AP-445

APPLICATION NOTE

# 8XC196KR Peripherals:

## A User's Point of View

ROB KOWALCZYK

STEVE McINTYRE

April 1992

Order Number: 270873-001

Contact your local Intel sales office or your distributor to obtain the latest specifications before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683

# 8XC196KR Peripherals: A User's Point of View

## CONTENTS



## CONTENTS

## CONTENTS PAGE



## CONTENTS PAGE

# Figures

## Programs

## 1.0 INTRODUCTION

High Speed Event control is a common occurrence in todays control applications. Also mixing analog and digital control in the same application is becoming a necessity.

In 1982 Intel introduced the first member of the 16-bit microcontroller family (MCS®-96): the 8096 device. This family has grown from that first introduction to todays 4th generation of highly integrated, 1 micron CHMOS technology members. The 8XC196KR, 8XC196KQ, 8XC196JR, and 8XC196JQ. (Known hereafter as 8XC196KR).

These devices combine high speed 16- and 32-bit precision calculation capability (100% instruction set compatible with the MCS-96 product family) with a dedicated I/O subsystem that has no equal. Figure 1-1 illustrates the complete functional blocks that make up the 8XC196KR devices.

This Ap-note will briefly describe the 8XC196KR CPU and peripherals with example applications for each.

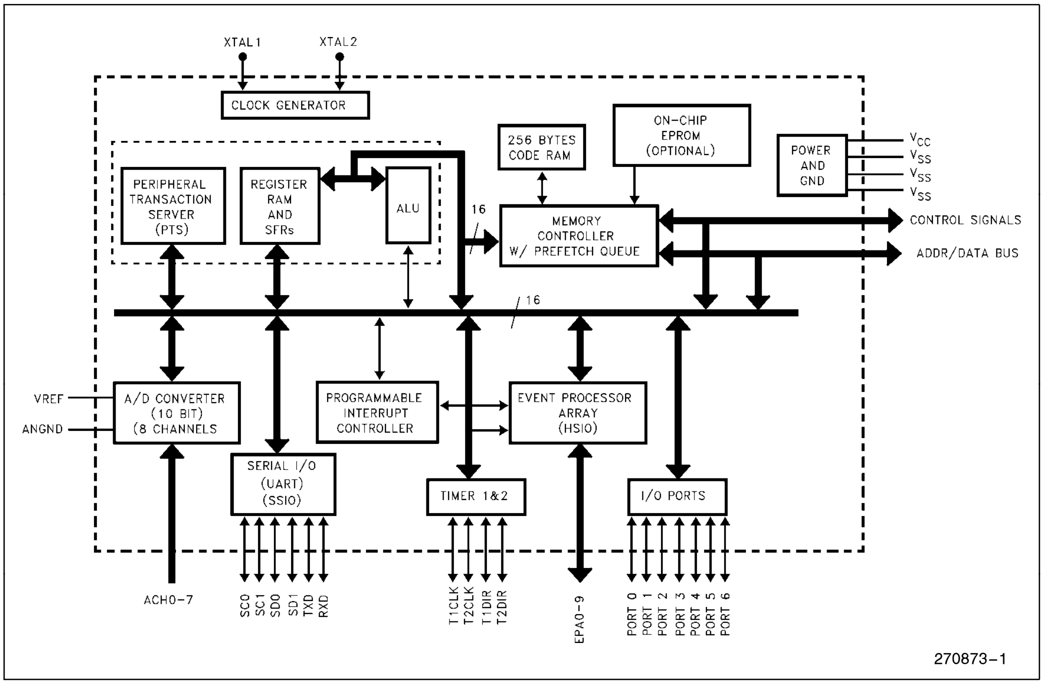



**Figure 1-1. 8XC196KR Block Diagram**

## 1.1 8XC196KR Overview

### 1.1.1 GENERAL DESCRIPTION – CPU

The 8XC196KR instructions are a true instruction super set of past 8096 devices (8X9XBH and 8XC196KB). It uses a 16-bit ALU which operates on 512 bytes of registers instead of an accumulator. Like the 8096, any location within the 512 byte register file can be used as source or destinations for most of the instruction addressing modes.

This register to register architecture is common to the MCS-96 family. Many of the instructions can operate on bytes, words and double words from anywhere in the 64K byte address space. To assist in the understanding of the 8XC196KR memory, a map is shown in Figure 1-2.

| Address | Type of Memory |
|---|---|
| 0FFFFh<br>06000h | External Memory |
| 05FFFh<br>02080h | Internal/External EPROM |
| 0207Fh<br>0205Eh | Internal/External EPROM<br>(Int. Vectors/Open/Reserved) |
| 0205Dh<br>02030h | Internal/External EPROM<br>(Int. Vectors) |
| 0202Fh<br>02020h | Internal/External EPROM<br>(Security Key) |
| 0201Fh<br>02014h | Internal/External EPROM<br>(CCB0/CCB1/Reserved) |
| 02013h<br>02000h | Internal/External EPROM<br>(Int. Vectors) |
| 01FFFh<br>01F00h | Internal SFR Registers |
| 01EFFh<br>00500h | External Memory |
| 004FFh<br>00400h | Internal Code RAM |
| 003FFh<br>00200h | External Memory |
| 001FFh<br>00018h | Internal Register RAM |
| 00017h<br>00000h | Internal Core SFR Registers |

**Figure 1-2. 8XC196KR Memory Map**

The lower 24 bytes of the register file contain Special Function Registers (SFRs) that are used to control on-chip peripherals (similar to past 8096 devices). In addition to these SFRs, the 8XC196KR device has 256 more SFRs located from 1F00H through 1FFFH. All RAM memory (Register memory from 0000H to 01FFH and Code RAM memory from 0400H to 04FFH) can be kept alive using the low current power-down or idle modes.

*Accessing RESERVED SFR/data memory locations are not allowed. Memory locations 1F00h through 1F5Fh are also considered RESERVED.*

Figures 1-3 and 1-4 show the layout and reset values of the SFRs on the 8XC196KR. Most of these registers are Read **and** Writable (unlike those in past 8096 devices).

| SFR | Location | RESET Value | R/W |
|---|---|---|---|
| POPIN | 1FDAH | XXH | R |
| P3PIN<br>P3REG | 1FFEH<br>1FFCH | XXH<br>0FFH | R<br>R/W |
| P4PIN<br>P4REG | 1FFFH<br>1FFDH | XXH<br>0FFH | R<br>R/W |
| P1PIN<br>P1REG<br>P1IO<br>P1SSEL | 1FD6H<br>1FD4H<br>1FD2H<br>1FD0H | XXH<br>0FFH<br>0FFH<br>00H | R<br>R/W<br>R/W<br>R/W |
| P2PIN<br>P2REG<br>P2IO<br>P2SSEL | 1FCFH<br>1FCDH<br>1FCBH<br>1FC9H | 1XXXXXXXB<br>7FH<br>7FH<br>80H | R<br>R/W<br>R/W<br>R/W |
| P6PIN<br>P6REG<br>P6IO<br>P6SSEL | 1FD7H<br>1FD5H<br>1FD3H<br>1FD1H | XXH<br>0FFH<br>0FFH<br>00H | R<br>R/W<br>R/W<br>R/W |
| P5PIN<br>P5REG<br>P5IO<br>P5SSEL | 1FF7H<br>1FF5H<br>1FF3H<br>1FF1H | 1XXXXXXXB<br>0FFH<br>0FFH<br>80H | R<br>R/W<br>R/W<br>R/W |
| INT_MASK<br>INT_MASK1<br>INT_PEND<br>INT_PEND1 | 0008H<br>0013H<br>0009H<br>0012H | 00H<br>00H<br>00H<br>00H | R/W<br>R/W<br>R/W<br>R/W |
| PTS_SRV<br>PTS_SELECT | 0006H<br>0004H | 00H<br>00H | R/W<br>R/W |
| WDT | 000AH | 00H | R |

**Figure 1-3. Special Function Registers**

| SFR | Location | RESET Value | R/W |
|---|---|---|---|
| USFR | 1FF6H | XXH | W |
| SLPCMD | 1FFAH | 00H | R/W |
| SLPSTAT | 1FF8H | 00H | R/W |
| SLPFUNREG | 1FFBH | 00H | R/W |
| EPAIPV | 1FA8H | 00H | R |
| EPA_MASK | 1FA0H | 0000H | R/W |
| EPA_PEND | 1FA2H | 0000H | R/W |
| EPA_MASK1 | 1FA4H | 00H | R/W |
| EPA_PEND1 | 1FA6H | 00H | R/W |
| TIMER1 | 1F9AH | 0000H | R/W |
| TIMER1_CONTROL | 1F98H | 00H | R/W |
| TIMER2 | 1F9EH | 0000H | R/W |
| TIMER2_CONTROL | 1F9CH | 00H | R/W |
| AD_TIME | 1FAFH | 0FFH | R/W |
| AD_TEST | 1FAEH | 0C0H | R/W |
| AD_COMMAND | 1FACH | 0C0H | R/W |
| AD_RESULT | 1FAAH | 07F80H | R/W |
| SP_BAUD | 1FBCH | 0000H | W |
| SP_CONTROL | 1FBBH | 0E0H | R/W |
| SP_STATUS | 1FB9H | 0BH | R/W |
| SBUF_TX | 1FBAH | 00H | R/W |
| SBUF_RX | 1FB8H | 00H | R/W |
| SSIO_BAUD | 1FB4H | 0XXXXXXXB | W |
| SSIO_STCR1 | 1FB3H | 00H | R/W |
| SSIO_STCR0 | 1FB1H | 00H | R/W |
| SSIO_STB1 | 1FB2H | 00H | R/W |
| SSIO_STB0 | 1FB0H | 00H | R/W |
| COMP_TIME1 | 1F8EH | XXXXH | R/W |
| COMP_TIME0 | 1F8AH | XXXXH | R/W |
| COMP_CONTROL1 | 1F8CH | 00H | R/W |
| COMP_CONTROL0 | 1F88H | 00H | R/W |
| EPA_TIME9 | 1F86H | XXXXH | R/W |
| EPA_TIME8 | 1F82H | XXXXH | R/W |
| EPA_TIME7 | 1F7EH | XXXXH | R/W |
| EPA_TIME6 | 1F7AH | XXXXH | R/W |
| EPA_TIME5 | 1F76H | XXXXH | R/W |
| EPA_TIME4 | 1F72H | XXXXH | R/W |
| EPA_TIME3 | 1F6EH | XXXXH | R/W |
| EPA_TIME2 | 1F6AH | XXXXH | R/W |
| EPA_TIME1 | 1F66H | XXXXH | R/W |
| EPA_TIME0 | 1F62H | XXXXH | R/W |
| EPA_CONTROL9 | 1F84H | 00H | R/W |
| EPA_CONTROL8 | 1F80H | 00H | R/W |
| EPA_CONTROL7 | 1F7CH | 00H | R/W |
| EPA_CONTROL6 | 1F78H | 00H | R/W |
| EPA_CONTROL5 | 1F74H | 00H | R/W |
| EPA_CONTROL4 | 1F70H | 00H | R/W |
| EPA_CONTROL3 | 1F6CH | 0FE00H | R/W |
| EPA_CONTROL2 | 1F68H | 00H | R/W |
| EPA_CONTROL1 | 1F64H | 0FE00H | R/W |
| EPA_CONTROL0 | 1F60H | 00H | R/W |

**Figure 1-4. Special Function Registers**

#### 1.1.2 INTEGRATED I/O SUBSYSTEM

Some of the I/O features on the 8XC196KR are similar to past 8096 devices. But, a great deal of the I/O and it's specific functions have changed for the better.

For example, the WatchDog Timer (WDT) is an internal timer which can be used to reset the system when software fails to operate properly. On past 8096 devices this feature was turned off until initially written. On the 8XC196KR devices, the Chip Configuration Byte (CCB1) contains a bit (bit 3) which can have this feature always enabled. Now if software fails **before** it gets to the WDT initialization code, it will reset the system.

The 8XC196KR device still contains an Analog to Digital converter, High Speed Input Capture and Output Compare called Event Processor Array (EPA), an integrated 16-bit timer/counter subsystem, and Asynchronous/Synchronous Serial I/O.

In addition to the above peripherals, the 8XC196KR device has an additional Synchronous Serial I/O port, an Additional timer/counter, faster interrupt response capability through the Peripheral Transaction Server (PTS), an 8 bit slave port that allows other CPUs in the system to request information from the 8XC196KR through interrupt control, and an additional 8 pins of I/O.

This integration of I/O, memory, ALU capability, and overall system speed makes the 8XC196KR device a perfect fit in such applications as: Motor Control, Engine Control, Anti-lock Brakes, Suspension Control, Hard Disk Drive Controllers, Printer Control, as well as many others.

### 1.2 New 8XC196KR Instructions

The 8XC196KR device is an instruction super set of the past 8096 devices (8X9XBH , 8XC196KB). The software used to Assemble / Compile / Link and Locate programs still holds true for the 8XC196KR. (ASM96, RL96, iC96, PL/M96, .....)

The 8XC196KR device has additional instructions not seen before by MCS-96 programmers. These 6 new instructions are EPTS (Enable PTS) / DPTS (Disable PTS), XCH (eXCHange word) / XCHB (eXCHange Byte), BMOVI (Interruptable Block MOVe), and TIJMP (Table Indirect JuMP).

**BMOVI** Interruptable Block Move has the same form and function as the BMOV instruction except the interrupt request status is checked after each move is completed. If an interrupt is pending and unmasked the block move operation is suspended and the interrupt service routine is invoked. Following the end of the interrupt service routine the block move continues. The BMOVI instruction, unlike the BMOV, will update the counter register, if interrupted.

Assembly Language:

| | PTRs | CNT |
|---|---|---|
| BMOVI | Lreg , | Wreg |

Object code:

| | | |
|---|---|---|
| 0CDH | <wreg> | <lreg> |

**DPTS** Disable PTS (Peripheral Transaction Server) by clearing the PSE flag in the PSW register.

Assembly Language:

DPTS

Object code:

0ECH

**EPTS** Enable PTS (Peripheral Transaction Server) by setting the PSE flag in the PSW register.

Assembly Language:

EPTS

Object code:

0EDH

**TIJMP** Table Indirect Jump, jumps to an address selected out of a table of addresses. This is a three operand instruction with one operand pointing to the base of the jump table, a second pointing indirectly to a 7-bit index value (0 to 128 decimal) and a third is an immediate operand (7-bit) which is used as a mask for the index value.

Assembly Language:

| | BASE INDEX | MASK |
|---|---|---|
| TIJMP | Wreg1,[Wreg2], | #Mask |

Object code:

| | | | |
|---|---|---|---|
| 0E2H | <wreg2> | <mask> | <wreg1> |

**XCH/XCHB** Exchange Word and Exchange Byte, exchanges the contents of two memory locations. The immediate and indirect addressing modes are NOT supported, only the direct and indexed (short and long).

Assembly Language:

| | DST | SRC |
|---|---|---|
| XCHB | Breg1, | Breg2 |
| XCHB | Breg1, | Offset [Wreg2] |
| XCH | Wreg1, | Wreg2 |
| XCH | Wreg1, | Offset [Wreg2] |

Object Code:

| | | | |
|---|---|---|---|
| 14H | Breg2 | Breg1 | |
| 1BH | Wreg2 | offset_low {offset_high} | breg1 |
| 04H | Wreg2 | Wreg1 | |
| 0BH | Wreg2 | offset_low {offset_high} | wreg1 |

For short indexed addressing modes the second offset byte is omitted from the object code stream. For long indexed addressing mode, both bytes of offset are required, making the instruction a 5 byte instruction.

### 1.2.1 52 LEAD DEVICES

Intel offers a 52 lead version of the 8XC196KR device: the 8XC196JR and 8XC196JQ devices. The first samples and production units use the 8XC196KR die and bond it out in a 52 lead package.

It is important to point out some functionality differences because of future devices or to remain software consistent with the 68 lead device. Because of the absence of pins on the 52 lead device some functions are not supported.

52 Lead Unsupported Functions:

Analog Channels 0 and 1.
INST pin functionality.
SLPINT pin support.
HLD#/HLDA# functionality.
External clocking/direction of Timer1.
WRH# or BHE functions.
Dynamic buswidth.
Dynamic wait state control.

The following is a list of recommended practices when using the 52 lead device:

(1) **External Memory.** Use an 8-bit bus mode only. There is neither a WRH# or BUSWIDTH pin. The bus can not dynamically switch from 8- to 16-bit or vice versa. Set the CCB bytes to an 8-bit only mode, using WR# function only.

(2) **Wait State Control.** Use the CCB bytes to configure the maximum number of wait states. If the READY pin is selected to be a system function, the device will lockup waiting for READY. If the $\overline{READY}$ pin is configured as LSIO (default after $\overline{RESET}$), the internal logic will receive a logic "0" level and insert the CCB defined number of wait states in the bus cycle. DON'T USE IRC = "111".

(3) **NMI support.** The NMI is not bonded out. Make the NMI vector at location 203Eh vector to a Return instruction. This is for glitch safety protection only.

(4) **Auto-Programming Mode.** The 52 lead device will ONLY support the 16-bit zero wait state bus during auto-programming.

(5) **EPA4 through EPA7.** Since the JR and JQ devices use the KR silicon, these functions are in the device, just not bonded out. A programmer can use these as compare only channels or for other functions like software timer, start and A/D, or reset timers.

(6) **Slave Port Support.** The Slave port can still be used on the 52 lead devices. The only function removed is the SLPINT output function.

(7) **Port Functions.** Some port pins have been removed. P5.7, P5.6, P5.5, P5.1, P6.2, P6.3, P1.4 through P1.7, P2.3, P2.5, P0.0 and P0.1. The PxREG, PxSSEL, and PxIO registers can still be updated and read. The programmer should not use the corresponding bits associated with the removed port pins to conditionally branch in software. Treat these bits as RESERVED.

Additionally, these port pins should be setup internally by software as follow:

1. Written to PxREG as "1" or "0".
2. Configured as Push/Pull, PxIO as "0".
3. Configured as LSIO.

This configuration will effectively strap the pin either high or low. *DO NOT Configure as Open Drain output "1", or as an Input pin. This device is CMOS.*

## 1.3 Windowing

The 8XC196KR contains 512 bytes of memory, located from 00h to 1FFh. An additional 256 bytes of on chip SFRs (Special Function Registers) located at 1F00h–1FFFh. Accesses directly to any location other than 00h–FFh would require a 16 bit address.

The 8XC196KR device has a mechanism known as vertical windowing which allows portions of 16-bit memory (0000–1FFh and 1F00h–1FFFh) to be remapped to an 8-bit address in the 0080h - 00FFh register RAM area.

Any address accesses using an 8-bit re-mapped address will be windowed through to the 16-bit address.

*The 8XC196KR core has the capability to add up to 1K of register RAM (0000–03FFh) and 1K of SFR space (1C00–1FFFh). However, only 512 bytes of register RAM is accessible along with 256 bytes of SFRs (locations 1F00–1FFFh). 1F00h through 1F5Fh is considered RESERVED. Only the non-RESERVED locations should be selected for windowing. Any attempt to window outside this area will result in reading of all 1's and writing to the bit bucket. In addition, the SFRs located from 1FE0h–1FFFh can NOT be accessed through any window. Any attempt to write these register through a window will have no effect on these SFRs; reading these registers through a window will result in FFh or FFFFh being read.*

Windows can be selected to be either 32, 64 or 128 bytes, and will be mapped into locations E0h- -FFh, C0h–FFh, or 80h–FFh, respectively. Control over the window is obtained through the use of the WSR register located at 14h. The bit map of the WSR depends somewhat on the size of the window. The MSB of the WSR is not used for windowing, but rather is used to control whether or not outside bus masters can request control of the external bus (HOLD/HLDA enabling). The next three bits either determine the size of the window, or, for 64 and 32 byte windows, part of the offset address.

Bits 6,5 and 4 determine which "window" size from memory is to be used. Bits 3,2,1, and 0 determine which block the device will window to.

For example, suppose the programmer wanted a window of 128 bytes. First, visualize the memory as being divided into consecutive blocks of 128 bytes each. Note that there is a gap from 400h– 1EFFh. This will make 15 blocks of 128 bytes that can be windowed through 80h–0FFh (skipping memory between 400h–1EFFh).

Number the blocks starting with zero. Furthermore, assume that the programmer wanted to window addresses from 180h–1FFh to 80–FFh. This is the third block of 128 bytes in memory. The fifteen block will be 1F80h–1FFFh (starting with 0000h). This means that the WSR should contain "001" in the upper bits 6,5, and 4, to select 128 byte windowing, and "0011" (3) in the lower nibble to select block number 3.

Figures 1-5, 1-6, and 1-7 illustrate all valid codes for the WSR (on the 8XC196KR), and the corresponding windows opened. Using other WSR values (other than "00") is not supported or recommended.

The WSR register is stacked on a PUSHA instruction. This will save the WSR value on the stack while executing an Interrupt Service Routine (ISR). A PUSHA instruction has no affect on the WSR register. Before returning from the ISR, a POPA will return the WSR to the previous value prior to entering the ISR.

| 128 Byte "Window" | | |
|---|---|---|
| **WSR** | **Window** | **Remapped** |
| x001 0000 | 0000 | 0080 to 00FF |
| x001 0001 | 0080 | |
| x001 0010 | 0100 | |
| x001 0011 | 0180 | |
| x001 1110 | 1F00 | |
| x001 1111 | 1F80 | |

**Figure 1-5. 128-Byte Windows**

| 64 Byte "Window" | | |
|---|---|---|
| **WSR** | **Window** | **Remapped** |
| x010 0000 | 0000 | 00C0 to 00FF |
| x010 0001 | 0040 | |
| x010 0010 | 0080 | |
| x010 0011 | 00C0 | |
| x010 0100 | 0100 | |
| x010 0101 | 0140 | |
| x010 0110 | 0180 | |
| x010 0111 | 01C0 | |
| x011 1101 | 1F40 | |
| x011 1110 | 1F80 | |
| x011 1111 | 1FC0 | |

**Figure 1-6. 64-Byte Windows**

| 32 Byte "Window" | | |
|---|---|---|
| **WSR** | **Window** | **Remapped** |
| x100 0000 | 0000 | 00E0 to 00FF |
| x100 0001 | 0020 | |
| x100 0010 | 0040 | |
| x100 0011 | 0060 | |
| x100 0100 | 0080 | |
| x100 0101 | 00A0 | |
| x100 0110 | 00C0 | |
| x100 0111 | 00E0 | |
| x100 1000 | 0100 | |
| x100 1001 | 0120 | |
| x100 1010 | 0140 | |
| x100 1011 | 0160 | |
| x100 1100 | 0180 | |
| x100 1101 | 01A0 | |
| x100 1110 | 01C0 | |
| x100 1111 | 01E0 | |
| x111 1011 | 1F60 | |
| x111 1100 | 1F80 | |
| x111 1101 | 1FA0 | |
| x111 1110 | 1FC0 | |
| x111 1111 | 1FE0 | |

**Figure 1-7. 32-Byte Windows**

### 1.3.1 EXAMPLES OF VERTICAL WINDOWS

To fully understand the windowing capability, some working examples are needed. Window 1Fh will be used; this remaps memory from 1F80h–1FFFh to 0080h–00FFh (00E0–00FFh have no effect on the SFRs because they will be windowed to memory mapped I/O locations 1FE0h through 1FFFh).

Assume the following code segment is executed without windowing. The results of this will be compared with the same code being executed with a window active.

```
(1)   ClrB   WSR
(2)   Ld     9Ah,#3000h
(3)   Ld     86h,9Ah
(4)   Ld     70h,86h[0]
(5)   Ld     72h,1F9Ah[0]
(6)   Ld     82h,[9Ah]
(7)   St     76h,0Ah[9Ah]
```

Three assumptions shall be made concerning the state of memory before the above code is executed: 1) Location 3000h contains 0303h, 2) location 76h contains 0A0Ah, and 3) the value in timer1 is 1111h (timer is not active). After the code is executed, the registers will be in the following state:

```
(1)   WSR:     00h
(2)   9Ah:     3000h
(3)   86h:     3000h
(4)   70h:     3000h
(5)   72h:     1111h
(6)   82h:     0303h
(7)   300Ah:   0A0Ah
```

These results are consistent with what would normally be expected without knowledge of windowing. Let the following code be executed with all of the above assumptions intact, but with windowing. Locations from 1F80h–1FFFh will be window through 0080h–00FFh.

```
(1)   LdB   WSR, #1Fh
(2)   Ld    9Ah,#1234h
(3)   Ld    86h,9Ah
(4)   Ld    70h,86h[0]
(5)   Ld    72h,1f9Ah[0]
(6)   Ld    82h,[9Ah]
(7)   St    76h,0Ah[9Ah]
```

Assume that location 1234h contains 0202h, 0080h contains 3000h, and 76h still contains 0A0Ah. The following results will be obtained:

```
(1)   WSR:     1Fh
(2)   1F9Ah:   1234h
(3)   1F86h:   1234h
(4)   70h:     3000h
(5)   72h:     1234h
(6)   1F82h:   0202h
(7)   123Eh:   0A0Ah
```

Contrast this with the results from the first code segment and note the following differences.

(1) The WSR is loaded with the windowed value that remaps 80h through 0FFh to 1F80h through 1FFFh.

(2) The immediate addressing mode moves immediate data through the window, into the 16-bit address of 1F9Ah, not 009Ah.

(3) Using the direct addressing mode, both the source (9Ah) and the destination (86h) are affected by the open window. This will move data from absolute address 1F9Ah (TIMER1) and place it in absolute address 1F86h (EPA_TIME9).

(4) Here the short indexed addressing mode is used to load register 70h from absolute location 80h + 00h. Notice that windowing does not affect any part of this example. Location 70h is not in the window. The indexed offset value (80h) is a constant and is NOT 1F80h. And lastly, register 0 (00) is not windowable.

(5) This is similar to example (4). Only the long indexed addressing mode is used. Here the 16-bit offset (1F9Ah) is added to the contents of the 00 register to get the address 1F9Ah. The contents of 1F9Ah is then stored in register 72h. No window affect.

(6) The indirect example has a great affect by windowing. Both the source and destination for this addressing mode refers to 8-bit registers. Hence, the contents of 1F9Ah is read and used as a pointer to a 16-bit address. That value is stored through the window to 1F82h (EPA_TIME8). If Auto-incrementing were used, the register 9Ah (1F9Ah through the window) would be incremented by 2.

(7) Lastly the Indexing mode example with the index register being affected by the window. Register 9Ah (1F9Ah through the window) is read and offset by immediate value #0Ah forming 123Eh. This address (123Eh) is the destination for register 76h (unaffected by the window).

As a final example, consider the following piece of code; try to determine exactly what it does before reading on.

```
(1)   LdB   WSR, #40h
(2)   OrB   0F3h, #18h
(3)   LdB   80h, #0E1h
(4)   LdB   0EAh, #1Eh
(5)   StB   80h, 0EAh
(6)   St    0E0h, 0F8h
(7)   LdB   0F4h, 0E0h
```

Which window was opened? The WSR was written a 0100000b, which selects a 32 byte window, starting at block 0. So 00h–1Fh was windowed to E0h–FFh. Next, INT_MASK1 was accessed through the window, and the Receive Interrupt (RI) and the Transmit Interrupt (TI) of the serial port were enabled.

*The next three instructions are dangerous as they enable the watchdog timer.*

The stack pointer is then set to 0000h and finally, the window is reset by accessing the window select register (WSR) through a window!

### 1.4 Top 5 Issues With Windowing

1. The PUSHA will NOT clear the WSR. But a POPA will restore the WSR from the stack.
2. Both source and destination bytes of the Direct Addressing Mode can be affected by the window selected.
3. The Offset in the Indexed Addressing Mode is NOT windowable.
4. The PTS uses all 16-bit addresses and is therefore unaffected by the window selected.
5. Without proper understanding of the window mechanism, the user can get into big trouble using windows. Beware the window!

## 2.0 INTERRUPTS AND THE PERIPHERAL TRANSACTION SERVER (PTS)

The PTS is a microcoded hardware interrupt service routine that "steals" bus cycles to execute. It is able to do a special encoded interrupt service routine in the time it takes to execute one instruction.

In a way, the PTS is a one instruction interrupt service routine that executes *without* stack or PSW being modified, and *with* minimum CPU overhead. A simple PTS cycle is only 1.875uS at 16 MHz (less than it takes to do a divide and just a 1 state time longer than a multiply).

There is a new bit in the PSW to control the global enable of PTS interrupts (PSE bit in the PSW). This bit is set using the EPTS instruction and disabled using the DPTS instruction which is discussed in the new instruction section of this document.

On past 8096 devices an interrupt requests sets the INT_PEND bit in the core, the core looks at the corresponding INT_MASK bit to see if it should "vector" to a software interrupt service routine. Also provided that the interrupts are enabled (I bit in the PSW). This type of interrupt response is still on the 8XC196KR devices.

The 8XC196KR core implements a small twist to those events. Before "vectoring" to the software interrupt routine, the core checks another bit: the corresponding PTS_SELECT bit (location 04H:WORD). If this bit is also set, the core will vector to a microcoded interrupt service routine INSTEAD of the software interrupt service routine (provided the PSE bit in the PSW is also set).

The PTS is able to perform a DMA-like response to any interrupt source. Figure 2-1 illustrates the location, priority and source for all of the PTS interrupt vectors available on the 8XC196KR.

*All PTS interrupts have higher priority than normal software interrupts. There is a vector for the EPAINTx interrupts, but it is NOT possible to do PTS cycles using this vector due to the nature of the INT_PEND bit (See the EPA section for details).*

There is no PTS vector for the NMI, TRAP, or Unimplemented Opcode. Also the PTS will not function while in Idle.

## 2.1 PTS Execution

As in normal software interrupt response, the current instruction is completed before the PTS interrupt cycle executes. The internal priority handler, handles the requests based on their priority. Next the PTS vector is read from the vector table to get the address of the PTS Control Block (PTSCB).

As with any instruction there are opcodes and operands. The PTSCB is no exception. It defines the "instruction" to execute when an interrupt request comes in. The first word is the "opcode". In some PTS modes the whole word is used, and in others it only uses the high byte of the word. (see individual PTS modes for details). The next three words are the operands of the PTS interrupt cycle.

*If the 8XC196KR device is running from external memory, this interrupt vector fetch may be the only evidence that a PTS cycle has executed.*

The CPU executes the proper PTS instruction based on the contents of the PTS Control Block (Moving data from one location to another, internal or external - doing the special PWM / PWM toggle - or A/D scan modes).

Instead the PTS executes a single microcoded instruction that resides in Register RAM (locations 0000H to 01FFH). Code RAM can not be used for PTS control blocks.

The PTSCB is a set of registers that defines how the PTS cycle is to be performed.

*This PTS control block or RAM registers are always on QUAD word boundaries (address = 0000000xxxxxx000). The assembler will not give an error message if this QUAD word boundary rule is violated. If the PTS vector points to a non-QUAD word boundary, upon execution of this PTS cycle, the CPU will round down to the nearest QUAD word boundary.*

| Number | Source | Vector Location | Priority |
|---|---|---|---|
| PTS15 | NMI - RESERVED | ——— | |
| INT15 | Non Maskable Interrupt | 203EH | 30 |
| PTS14 | EXTINT | 205CH | 29 |
| PTS13 | reserved | 205AH | 28 |
| PTS12 | RI | 2058H | 27 |
| PTS11 | TI | 2056H | 26 |
| PTS10 | XFR1 | 2054H | 25 |
| PTS09 | XFR0 | 2052H | 24 |
| PTS08 | CBF | 2050H | 23 |
| PTS07 | IBF | 204EH | 22 |
| PTS06 | OBE | 204CH | 21 |
| PTS05 | A/D Done | 204AH | 20 |
| PTS04 | EPAINT0 | 2048H | 19 |
| PTS03 | EPAINT1 | 2046H | 18 |
| PTS02 | EPAINT2 | 2044H | 17 |
| PTS01 | EPAINT3 | 2042H | 16 |
| PTS00 | EPAINTX (RESERVED) | 2040H | 15 |
| INT14 | EXTINT Pin | 203CH | 14 |
| INT13 | RESERVED | 203AH | 13 |
| INT12 | Receive SIO Interrupt | 2038H | 12 |
| INT11 | Transmit SIO Interrupt | 2036H | 11 |
| INT10 | SSIO channel 1 transfer | 2034H | 10 |
| INT09 | SSIO channel 0 transfer | 2032H | 9 |
| INT08 | Command Buffer Full SLP | 2030H | 8 |
| SPECIAL | Illegal Opcode | 2012H | N/A |
| SPECIAL | TRAP instruction | 2010H | N/A |
| INT07 | Input Buffer Full | 200EH | 7 |
| INT06 | Output Buffer Empty | 200CH | 6 |
| INT05 | A/D Complete | 200AH | 5 |
| INT04 | EPAINT0 | 2008H | 4 |
| INT03 | EPAINT1 | 2006H | 3 |
| INT02 | EPAINT2 | 2004H | 2 |
| INT01 | EPAINT3 | 2002H | 1 |
| INT00 | EPAINTX | 2000H | 0 |

**Figure 2-1. 8XC196KR Interrupt Priorities**

| | | | | | | |
|---|---|---|---|---|---|---|
| | UNUSED | UNUSED | UNUSED | UNUSED | CONST2(HI) | OPERAND #3 |
| | UNUSED | PTS_BURST | UNUSED | UNUSED | CONST2(LO) | |
| | PTS_DEST(HI) | PTS_DEST(HI) | REG (HI) | CONST1(HI) | CONST1(HI) | OPERAND #2 |
| | PTS_DEST(LO) | PTS_DEST(LO) | REG (LO) | CONST1(LO) | CONST1(LO) | |
| | PTS_SOURCE(HI) | PTS_SOURCE(HI) | S/D (HI) | PTS_SOURCE(HI) | PTS_SOURCE(HI) | OPERAND #1 |
| | PTS_SOURCE(LO) | PTS_SOURCE(LO) | S/D (LO) | PTS_SOURCE(LO) | PTS_SOURCE(LO) | |
| | PTS_CONTROL | PTS_CONTROL | PTS_CONTROL | PTS_CONTR OL | PTS_CONTROL | OPCODE |
| PTSVEC → | PTSCOUNT | PTSCOUNT | PTSCOUNT | UNUSED | UNUSED | |
| | **Single Transfer** | **Block Transfer** | **A/D Mode** | **PWM Mode** | **PWM Toggle** | |

**Figure 2-2. PTS Control Blocks (PTSCB)**

Figure 2-2 shows the 5 PTS modes available on the 8XC196KR devices. The bytes in the PTSCB labeled "UNUSED" can be used by the users' program as register RAM space. The PTS does not require information from these UNUSED locations and the data in the UNUSED locations will not be altered in any way.

## 2.2 PTS Modes

The PTSCB defines the *mode* or type of PTS cycle to perform when the interrupt request comes in. Five modes are provided on the 8XC196KR: a Single Transfer Mode, a Block Transfer Mode, an A/D Scan Mode, and two PWM Modes.

Any of these modes can be used for ANY interrupt source associated with the PTS vectors (except EPAINTx). ie: The A/D Scan was specifically designed to function with the A/D peripheral, but if the user can think of an application where the A/D scan mode would be used with the SIO peripheral, or any other peripheral it can be done.

### 2.2.1 SINGLE TRANSFER MODE

The Single Transfer Mode of PTS cycle can transfer data from any address to any other address in memory. The data can be a word or a byte, and the source and/or destination pointers can be optionally incremented.

This PTS mode uses six bytes out of the eight byte in the PTS Control Block. *The other 2 bytes can be used as regular scratch pad register. The PTS cycle has NO EFFECT on these two registers.*

Below is the PTS Control Byte description for the Single Transfer Mode.

**PTS_CONTROL**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| M2 | M1 | M0 | B/W | SU | DU | SI | DI |
| 1 | 0 | 0 | X | X | X | X | X |

**M0**
**M1** } 100 PTS Single Transfer Mode Select Bits
**M2**

**B/W** Byte (1)/Word (0) Transfer

**SU** Update PTS_SOURCE at the end-of-PTS

**DU** Update PTS_DEST at the end-of-PTS

**SI** PTS_SOURCE auto increment

**DI** PTS_DEST auto increment

**Figure 2-3. PTS Control Single Transfer**

The PTS Vector points to the first byte in the block. That byte is the COUNT register. The COUNT (PTS_COUNT) holds the number of PTS cycles to be performed before a normal software interrupt is called.

The PTS_COUNT register contains a value that is decremented at the *end* of the PTS cycle. When equal to zero the PTS_SELECT bit is cleared and the PTS_SRV bit is set. This causes a normal software interrupt routine to execute following the last PTS cycle.

The next byte (PTS Vector + 1) contains the PTS Control (PTS_CONTROL). This byte is present for all modes of operation. It defines the PTS mode, whether the PTS cycle is a word or byte transfer, and if the source and/or destination pointers should be incremented and/or updated during and at the end of the PTS cycle.

Figure 2-3 is an illustration of the PTS_CONTROL register for the Single Transfer Mode.

Notice that there is a bit for UPDATE and a bit for INCREMENT. Typically these are used in the Block Transfer Mode. The source and/or destination pointer is incremented if the bit is set. This increment happens after the PTS transfers the single byte or word to the destination pointer address.

The UPDATE bit in the control byte is set if the newly formed address (created by the increment function) is to be placed in the PTS control block source and destination pointer words.

In the Single transfer mode it makes no sense to increment without updating the pointers, or vice versa.

The Next word in the PTSCB is a Source Pointer. This word points to a 16-bit address (anywhere in memory). It can be an SFR location, or point to off chip memory.

If the PTS cycle directs data moves to read or write to external memory, an external bus cycle will be evident externally. If the PTS cycle is directed internally, no external evidence, other than the PTS vector fetch.

The last used word in the PTSCB is the Destination Pointer. It too points to ANYWHERE in the 64K address space.

### 2.2.2 SINGLE TRANSFER MODE EXAMPLE

Suppose that there is a 30 character message that needs to be transmitted out the Serial I/O port. The PTS can be setup to execute every time a transmit (TI) interrupt is requested.

The program would setup the serial port according to the application. The PTS vector at location 2056H would point to a QUAD word in the register RAM (IE: 01F8H) containing the PTSCB. The Source pointer in the PTSCB would point to the beginning message byte plus one (the first byte is sent manually). The Destination pointer in the PTSCB would point to the SBUF_TX special function register (location 1FBAH).

The PTS_COUNT would equal the number of byte to be transmitted minus one (the one that is send manually to start the SIO going).

The PTS_CONTROL is set for "Single transfer mode", Byte, Source is incremented and updated, and the destination is NOT increment or updated. (i.e., PTS_CTRL = 10011010B).

| | |
|---|---|
| PTS_COUNT | = (30 − 1) = 29 |
| PTS_CONTROL | = 9AH |
| PTS_SOURCE | = Buffer + 1 |
| PTS_DEST | = SBUF_TX (1FBAh) |

The main line program set up the PTS, Port 2, INT_MASK1 bit for the TI interrupt, and the PTS_SELECT bit for the TI interrupt is also set. Lastly the Interrupts and PTS Interrupts are enabled through the EI and EPTS instructions.

To get the ball rolling the first character is sent to the SIO (SFR register SBUF_TX). When the first character is sent over the serial port, the TI interrupt requests a PTS cycle. The PTS cycle will read the next byte from the message buffer and write it to the SBUF_TX register. The Source pointer is incremented to point to the next byte of the message.

This continues until the PTS_COUNT is decremented to zero. When this happens, the PTS_SELECT bit is reset ("0"), and the PTS_SRV bit is set. This indicates to the core that all the PTS cycles have been serviced and need to be setup again. As soon as the PTS_SRV bit is set a normal software interrupt service routine for the TI interrupt is executed (depending on the interrupt priorities and pending interrupts).

This example is shown below.

### 2.2.3 BLOCK TRANSFER MODE

The block transfer mode is very similar to the single transfer mode. It too transfers data from memory to memory on interrupt requests. The difference is the number of transfers performed for each interrupt request.

The block transfer mode is able to transfer upto 32 bytes or words for each interrupt request. *Since the PTS cycle cannot be interrupted, it is possible to have long latency times when using the block transfer mode. For example: if a block transfer mode is transferring 32 words from an external memory location to another external memory location, the latency could be as much as 500 to 600 states times.*

Like the single transfer mode, the block transfer mode has a PTS_COUNT register (works identical to the single transfer mode).

PTS_SOURCE and PTS_DEST pointers that point to the source and destination addresses for the transfer. And a PTS_CTRL that identifies the PTS mode, word/byte, increment and/or update pointer bits.

An additional byte (PTS_BURST) contains the number of transfers to perform for each interrupt request. The maximum number of transfers per interrupt request is 32.

```
$debug
$nolist
$include (kr.inc)
$list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  This program demonstrates the use of the serial port in
;  in mode 1.  The PTS is used to send data.
;  This program sends 30 bytes of dat out the serial port
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    windows for SRFs
;;;        1f
P2SSEL_W                EQU     0C9h:byte       ; window to 1fc9
P2REG_W                 EQU     0CDh:byte       ; window to 1fcd
P2IO_W                  EQU     0CBh:byte       ; window to 1fcb
SP_BAUD_W               EQU     0BCh:word       ; window to 1fbc
SP_CONTROL_W            EQU     0BBh:byte       ; window to 1fbb
SP_STATUS_W             EQU     0B9h:byte       ; window to 1fb9
SBUF_TX_W               EQU     0BAh:byte       ; window to 1fba

        CSEG    AT      2036h   ;sets up pointer to TI interrupt service
        dcw             TI_ISR
        CSEG    AT      2056h   ;pointer to PTS control block for TI
        dcw             TI_PTS_CNT

Temp    EQU     30H

        RSEG    AT      0E0H    ;PTS transmit control Block
TI_PTS_CNT:     DSB     1       ; Transmit Interrupt Count register
TI_PTS_CON:     DSB     1       ; Control register
TI_PTS_SRC:     DSW     1       ; Source pointer
TI_PTS_DST:     DSW     1       ; Destination pointer

        CSEG    AT      2080H
TI_START:
;  Set up port 2 and serial port control
        LdB     WSR, #1FH                       ; Open window 1FH
        Ld      SP,#0500h                       ; init stack
        ldb     SP_CONTROL_W, #01H              ; SIO mode 1 enable receiver and parity
        ld      SP_BAUD_W, #8067H               ; baud rate = 9600 at 16MHz
        orb     P2REG_W, #02h
        orb     P2SSEL_W, #03h                  ; Select TxD and RxD peripheral
        ldb     P2IO_W, #02H                    ;  RxD = Input
        ClrB    WSR
;set up PTS transmit control block
        Ldb     TI_PTS_CNT, #(Buffer_end-Buffer-1)   ; load count
        Ldb     TI_PTS_CON, #10011010B               ; single byte transfer mode
        Ld      TI_PTS_SRC, #Buffer                  ; beginning of the buffer
        Ld      TI_PTS_DST, #SBUF_TX                 ; point to SBUF_TX

        LdB     INT_MASK1, #00001000B                ; Enable TI interrupts
        Ld      PTS_SELECT, #0800H                   ; Enable TI of PTS

        LdB     Temp, [TI_PTS_SRC]+
        StB     Temp, SBUF_TX                   ; start transmission
        EPTS                                    ; enable PTS
        EI                                      ; enable interrupts
; The PTS will now handle the transmission of charecters.
; When a PTS routine is done, the  TI interrupt rountine below
; will be called.  This just notes that the PTS routine is finished.
;
wait:           Br      Wait
;
; There is a bug with the interrupt server on the 196.  It is possible for
;the wrong routine to be called.  During the latency period for a normal
;interrupt, if a PTS interrupt occurs, the normal interrupt routine for the
;PTS interrupt will be erroneously executed.
```

270873–16

**Program 1a. Send 30 bytes over the SIO using the PTS in Single Xfer Mode**

```
;
; Due to this bug, it is possible to enter these interrupt service
;routines before the PTS transfer is complete.  Therefore it is necessary
;to check to see if the PTS_SELECT bit for a given routine is set.  If
;it is, a PTS call should have been made, so the routine aborts and
;forces the call be setting the proper INT_PEND bit.
TI_ISR:
        PUSHA
        JBS    PTS_SELECT+1,3,abort1
        POPA
        RET
abort1:
        ORB    INT_PEND1,#08h        ; force call to PTS to correct bug
        POPA
        RET

Buffer:
        DCB      'This is a test of the PTS.' ; 26
        DCB      ' -' ; 29
Buffer_end:
        DCB      '-'
end
```

270873–17

**Program 1b. Send 30 bytes over the SIO using the PTS in Single Xfer Mode**

**PTS__CONTROL**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| M2 | M1 | M0 | B/W | SU | DU | SI | DI |
| 0 | 0 | 0 | X | X | X | X | X |

| | |
|---|---|
| **M0** **M1** **M2** | 000 PTS Block Transfer Mode Select Bits |
| **B/W** | Byte (1)/Word (0) Transfer |
| **SU** | Update PTS__SOURCE at the end-of-PTS |
| **DU** | Update PTS__DEST at the end-of-PTS |
| **SI** | PTS__SOURCE auto increment |
| **DI** | PTS__DEST auto increment |

**Figure 2-4. PTS Control Block Transfer**

### 2.2.4 BLOCK TRANSFER MODE EXAMPLE

Suppose that each time an EXTINT rising edge interrupt happens, a set of 8 word registers is to be initialized with data from an EPROM table.

The code would initialize the EXTINT pin as special function/input, setup the PTS Control Block, and set the INT__MASK1, PTS__SELECT, EI, and EPTS bits to perform a PTS cycle for each EXTINT request.

The PTS Control Block would be as follows:

The PTS__BURST equals 8 (for the 8 word registers that need to be initialized each EXTINT request), The PTS__COUNT is set to the number of times before a software interrupt service routine would occur, in this case: 5 (there are 5 different tables in EPROM to be loaded into the same set of registers). The PTS__SOURCE contains the address of the first word EPROM location to be read, The PTS__DEST contains the address of the first word register to be initialized.

The PTS__CTRL byte contains "00001011". With this control, the block transfer mode is selected, it is set to transfer WORDs instead of bytes, the source pointer is both incremented and updated, while the destination pointer is only incremented not updated.

When the EXTINT request comes in, the PTS will transfer data from the EPROM table of 8 words to the 8 word register. Between each transfer, both the source and destination pointers are incremented.

When the PTS block transfer mode is complete, the PTS__COUNT is decremented and the new source address created by the PTS is placed in the PTS__SOURCE pointer (updated) and is pointing to: (EPROM address + 8). But, the destination address formed by the PTS is NOT placed in the PTS__DEST location. The PTS__DEST pointer remains pointing at the first word of the 8 registers.

When the next EXTINT request comes in, the next block of 8 words from the EPROM table is written to the 8 word registers.

The coded example is shown on the following page.

```
$nolist
$include (macro.kr)
$list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  This program demonstrates the use of the EXTINT being used ;
;  with the PTS.  Each time an EXTINT rising edge interrupts  ;
;  the KR device, an 8 word block transfer PTS cycle initial- ;
;  izes and 8 word register bank.                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   windows for SRFs
;;;        1f
P2SSEL_W      EQU   0C9h:byte          ; window to 1fc9
P2REG_W       EQU   0CDh:byte          ; window to 1fcd
P2IO_W        EQU   0CBh:byte          ; window to 1fcb
        CSEG  AT    203CH          ;sets up pointer to EXTINT interrupt Service
        dcw         EXTINT_ISR
        CSEG  AT 205Ch             ;pointer to PTS control block for EXTINT
        dcw         EXTINT_PTS_CNT
        RSEG  AT    0E0H  ;PTS transmit control Block
EXTINT_PTS_CNT:   DSB   1               ; Count register
EXTINT_PTS_Con:   DSB   1               ; Control register
EXTINT_PTS_SRC:   DSW   1               ; Source pointer
EXTINT_PTS_DST:   DSW   1               ; Destination pointer
EXTINT_PTS_BST:   DSB   1               ; Burst Count
        DSEG  AT    400H
REGS: DSW   8                      ; 8 word registers that will get the data
        CSEG  AT    2080H
EXTINT_START:
;  Set up port 2
        LdB   WSR, #1FH                 ; Open window 1FH
        Ld    SP,#0500h                 ; init stack
        orb   P2REG_W, #00000100B       ; write out a 1 (open drain / input)
        orb   P2SSEL_W, #04H            ; Select EXTINT peripheral
        orb   P2IO_W, #04H              ;  RxD = Input
        ClrB  WSR
;set up PTS control block
        Ldb   EXTINT_PTS_CNT, #5        ;  load  count  with    5    characters
        Ldb   EXTINT_PTS_CON, #0Bh      ; Block/word/SU/SI/DI
        Ld    EXTINT_PTS_SRC, #TABLE    ; point to the beginning of the table
        Ld    EXTINT_PTS_DST, #REGS     ; point to 8 word registers
        LdB   EXTINT_PTS_BST, #8        ; 8 word transfer per EXTINT request
;set up the interrupts
        LdB   INT_MASK1, #01000000B     ; Enable EXTINT interrupt
        Ld    PTS_SELECT, #4000H        ; Enable EXTINT of PTS
        EPTS                       ; enable PTS
        EI                         ; enable interrupts
```

270873–18

**Program 2a. Using the EXTINT with the PTS Block Transfer Mode**

```
; The PTS will now handle the transfer of information from 5 different
; EPROM tables, into 8 registers in the Code RAM.  When the 5th one is loaded
; the PTS will reset the PTS_SELECT bit 14, and set the PTS_SRV bit 14, this
; will immediately execute a software EXTINT interrupt service routine.
; At this time the EPROM table pointer will be reset to TABLE and the Count
; will be reset to 5, and the PTS_SELECT bit 14 will be set again.
;
wait:
              Br      Wait
;
; There is a bug with the interrupt server on the 196.  It is possible for
;the wrong routine to be called.  During the latency period for a normal
;interrupt, if a PTS interrupt occurs, the normal interrupt routine for the
;PTS interrupt will be erroneously executed.
;
; Due to this bug, it is possible to enter these interrupt service
;routines before the PTS transfer is complete.  Therefore it is necessary
;to check to see if the PTS_SELECT bit for a given routine is set.  If
;it is, a PTS call should have been made, so the routine aborts and
;forces the call be setting the proper INT_PEND bit.
EXTINT_ISR:
       PUSHA
       JBS    PTS_SELECT+1,6,ABORT_EXTINT
; Legal EXTINT interrupt service routine call - reset PTSCB.
       Ldb    EXTINT_PTS_CNT, #5        ; load count with 5
       Ld     EXTINT_PTS_SRC, #TABLE ; point to the beginning of the table
       Or     PTS_SELECT, #4000H        ; re-enable EXTINT to PTS
       POPA
       RET
; If here, BUG occured
ABORT_EXTINT:
       ORB    INT_PEND1,#40h     ; reset PEND bit
       POPA                      ; and return
       RET
Table:
       DCW  1111H,1111H,1111H,1111H,1111H,1111H,1111H,1111H
       DCW  2222H,2222H,2222H,2222H,2222H,2222H,2222H,2222H
       DCW  3333H,3333H,3333H,3333H,3333H,3333H,3333H,3333H
       DCW  4444H,4444H,4444H,4444H,4444H,4444H,4444H,4444H
       DCW  5555H,5555H,5555H,5555H,5555H,5555H,5555H,5555H
end
```

270873–19

**Program 2b. Using the EXTINT with the PTS Block Transfer Mode**

#### 2.2.5 A/D SCAN MODE, PWM MODE AND PWM TOGGLE MODE

There are three other special modes of the PTS. Each are designed to work with specific peripherals on the 8XC196KR devices, but can be used with any interrupt source in the PTS_SELECT (except EPAINTx).

Each of these modes: A/D Scan Mode, PWM Toggle Mode and PWM Mode will be described later in this document. The A/D Scan Mode will be described in the A/D section and the PWM Toggle and PWM Modes in the EPA section.

## 2.3 PTS Latency Times

Since the PTS is simply an interrupt routine handled in microcode, it too has latency associated with its execution. As with normal software interrupts, the PTS has to wait until the current instruction has been processed before executing. The longest latency comes in within 4 states of the next instruction to be executed and the next instruction is a NORML (assuming that the PTS is enabled, and no non-interruptable block transfer PTS or BMOV instruction is to be executed).

That latency time is 43 state times.

The PTS cycle will be performed following the NORML instruction. Documented in Figure 2- 5 is the amount of states required to perform the PTS cycle. This time INCLUDES the vector to the PTS Control Block.

## 2.4 Top 5 Issues with PTS

(1) Make sure that the PTS control block is on QUAD word boundaries.

(2) The PTS can not be used with the EPAINTx interrupt vector.

(3) The PTS control block does not use windows (16 bit addresses only).

(4) Setting up the PTS:

1. Initialize PTS vector
2. Initialize PTS Control Block
3. Set PTS_SELECT bit
4. Enable PTS

(5) Beware of any anomolies on A-step silicon (Read 8XC196KR Erratas Carefully).

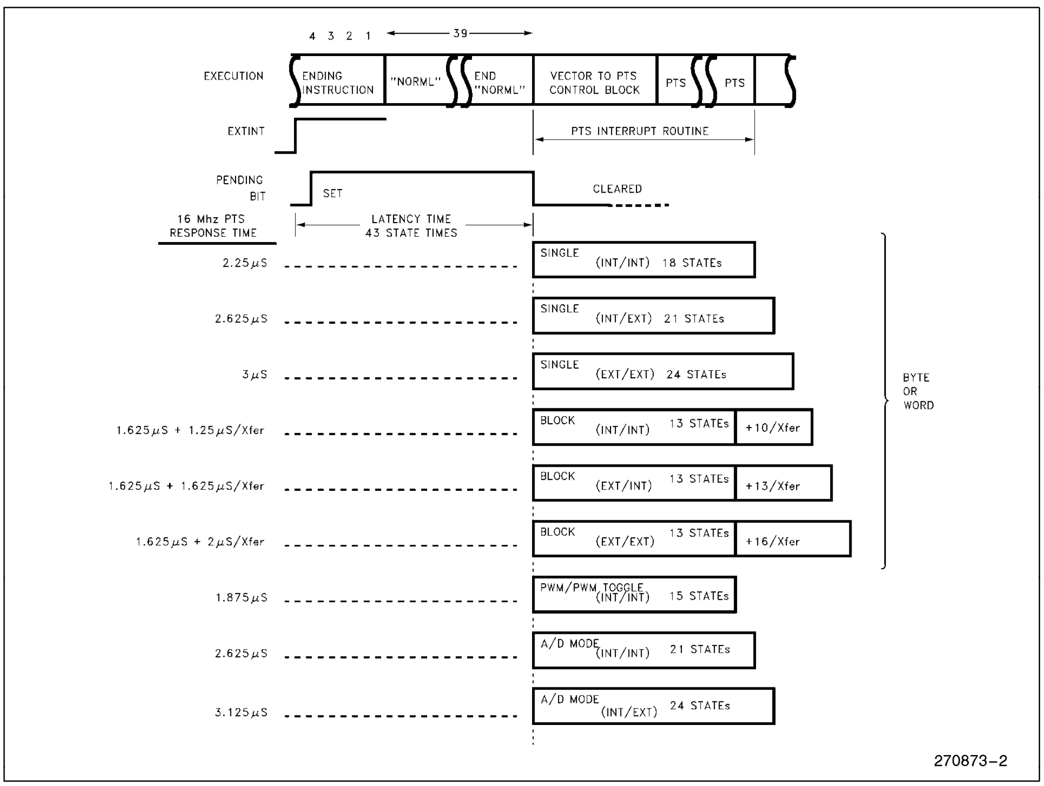
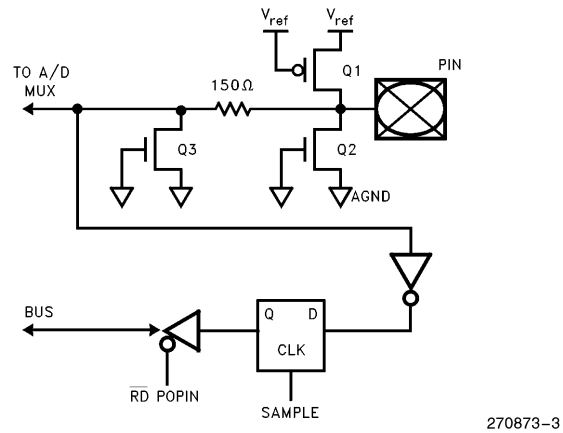



**Figure 2-5. PTS Interrupt Response Time**




**NOTE:**
Q1, Q2 and Q3 are ESD Protection Devices

**Figure 3-1. Input Port 0 Structure**

## 3.0 UNDERSTANDING THE PORTS

The 87C196KR/KQ/JR/JQ devices all have bidirectional ports that double as special function peripheral ports (EPA, SIO, SSIO, A/D, etc.). When the device is reset, most of these ports (P2.7 is an exception) are configured as Low Speed I/O, Open Drain output, with a weak pull up. In order to use these ports as their special function, they MUST be configured.

All the ports (except Port 0) have the same internal design. Some of the signals that drive the port cell are different. Below is a discussion about configuring and using these ports as either LSIO or Special Function Peripheral.

### 3.1 Port 0

The analog input pins on the 8XC196KR device are also called the Port 0 pins. These pins are input only. The input structure is shown in Figure 3-1.

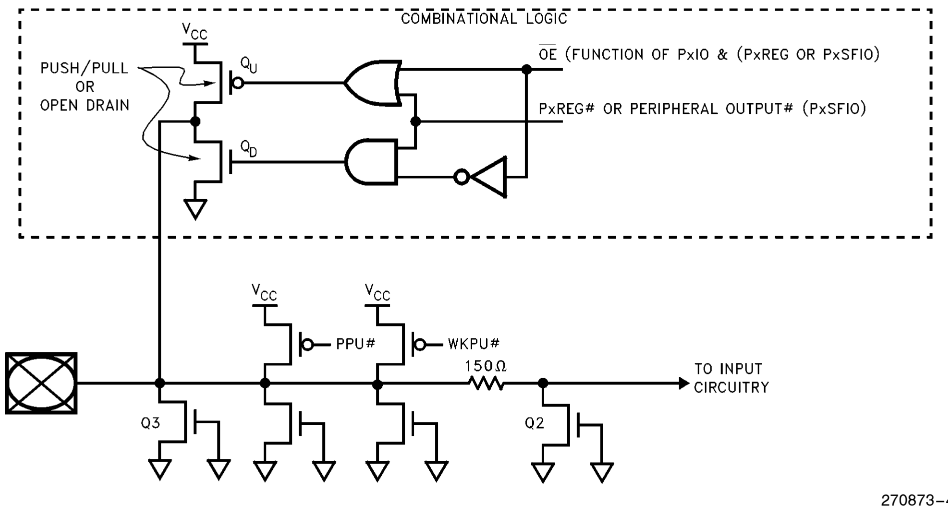
These port pins are input only and do not need to be configured if using them as analog input pins or digital inputs.

The input pins are sampled on Phase 1 and read into the bus on Phase 2. They have internal voltage clamping devices (ESD) as well as a 150 Ohm series poly resister. (See Figure 3-1).

Because of the way the multiplexer is designed (for minimum A/D errors), the maximum input current on any analog input is 1 mA. With this spec the A/D will yield an additional error on adjacent channels. Example: Force 1mA on channel 4; Channels 3 and 5 will have additional LSB errors.

Digital inputs are read through the P0PIN register (BYTE location 1FDAH). The A/D section of this document discusses the special function A/D peripheral.

This port should be straight forward, as it exists on all prior MCS®-96 devices.




* Q2 and Q3 and 150 Ohms Poly Resistor is for Input Protection.

**Figure 3-2. Ports 1, 2, 5 & 6 (and 3 / 4 - see notes)**

## 3.2 Port 1 / 2 / 6

These port pins are much different from the "quasi-bidirectional" ports seen on prior MCS®-96 devices. These are NOT "quasi-bidirectional" port pins. They do however, have some traits of quasi-bidirectional. (See Figure 3-2).

These pins have Schmitt trigger CHMOS inputs (with about 100mV of hysteresis and a $V_{IL} = 0.3V_{CC}$ and $V_{IH} = 0.7V_{CC}$) and CHMOS Outputs configurable as Open Drain or Push/Pull.

After the FALLING edge of $\overline{\text{RESET}}$ the signal $\overline{\text{PPU}}$ is active. The duration of the $\overline{\text{PPU}}$ is controlled by an R-C network that varies in width for different temperatures and process files, but it is at least 100 nS long. The transistor associated with the $\overline{\text{PPU}}$ signal is about 5-24K Ohms ($\sim$ 1mA / 5V drop). This pullup is used to charge pin loads before coming out of reset.

The active low $\overline{\text{RESET}}$ signal, will activate the $\overline{\text{WKPU}}$ signal. *This signal stays active till the user program writes to the PxSSEL register associated with each pin ("x" stands for 1, 2, or 6).*

The $\sim$ 150K Ω pullup that was present on the quasi-bidirectional ports is also present on the KR ports, but it can ONLY be turned off after RESET by writing to the PxSSEL register. The write to the PxSSEL register does the actual turning off of the $\overline{\text{WKPU}}$ signal, the data written makes no difference. Subsequent writes to the PxSSEL have no affect on the $\overline{\text{WKPU}}$ signal. On the next $\overline{\text{RESET}}$ low signal the $\overline{\text{WKPU}}$ signal will be turned on again.

This also means that the 1 to 0 input switching currents are at their worst case condition (50uA max) when the $\overline{\text{WKPU}}$ signal is active. If the pullup is turned off, there basically is NO/Little switching current.

| | PORT 1, 2, and 6 Truth Table | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| PxIO | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| PxREG | 0 | 1 | 0 | 1 | X | X | X | X |
| PxSFIO | X | X | X | X | 0 | 1 | 0 | 1 |
| PxSSEL | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $Q_U$ | off | on | off | off | off | on | off | off |
| $Q_D$ | on | off | on | off | on | off | on | off |
| PxPIN | Low | High | Low | HZ* | Low | High | Low | HZ |
| Port Config. | Push/Pull | | Open Drain | | Push/Pull | | Open Drain | |
| Port Function | LSIO | | | | Special Function | | | |

* During RESET and until first write to PxSSEL, $\overline{\text{WKPU}}$ is active.

**Figure 3-3. Port 1, 2, and 6 Truth Table**

The combinational logic is used to define the output. After writing to the PxREG register and the PxIO register, the port is configured as either PUSH/PULL or OPEN DRAIN (Provided the internal weak pullup was turned off by writing to the PxSSEL register). Figure 3-3 is a truth table for the combinational logic.

The PxSFIO register is an internal register that the special function peripheral writes to control the port. It is not visible to the core or the user. It's contents are seen at the port pin if the port is configured for special function (PxSSEL = "1") and output (Push/Pull or Open Drain).

After RESET the PxIO register is = "FFH" (Open Drain, Input). The PxREG is set to a "FFH" and the PxSSEL = LSIO. Even though the user accepts this as the port conditions, it is recommended that he at least configure the PxSSEL register after RESET in order to turn off the $\overline{\text{WKPU}}$ signal.

For example: If PORT 1 is configured as Low Speed Input port; write to the P1REG register ("FFH"), then write the P1IO register ("FFH" for open drain, Input), then finally write to the P1SSEL register ("00H" to select LSIO, and turn off internal $\overline{\text{WKPU}}$ signal).

Figure 3-4 is a list of all the special function registers associated with ports 0, 1, 2, 3, 4, and 6 with their respective absolute locations. These registers can be "windowed" via the WSR register in the core.

| SFR | Location | RESET Value |
|---|---|---|
| POPIN | 1FDAH | XXH |
| P3PIN | 1FFEH | XXH |
| P3REG | 1FFCH | 0FFH |
| P4PIN | 1FFFH | XXH |
| P4REG | 1FFDH | 0FFH |
| P1PIN | 1FD6H | XXH |
| P1REG | 1FD4H | 0FFH |
| P1IO | 1FD2H | 0FFH |
| P1SSEL | 1FD0H | 00H |
| P2PIN | 1FCFH | 1XXXXXXXB |
| P2REG | 1FCDH | 7FH |
| P2IO | 1FCBH | 7FH |
| P2SSEL | 1FC9H | 80H |
| P6PIN | 1FD7H | XXH |
| P6REG | 1FD5H | 0FFH |
| P6IO | 1FD3H | 0FFH |
| P6SSEL | 1FD1H | 00H |
| P5PIN | 1FF7H | 1XXXXXXXB |
| P5REG | 1FF5H | 0FFH |
| P5IO | 1FF3H | 0FFH |
| P5SSEL | 1FF1H | 80H |

**Figure 3-4. Port Reset Values**

## 3.3 Port 3 / 4

On past MCS®-96 devices, these ports have been open-drain. This is still true for the KR device ports 3 and 4. They are the Address and Data bus as well as open drain input/output port pins.

The structure of this port is similar in layout to the structure of the other ports 1, 2, and 6. The differences lie in the inputs to the port circuitry ($\overline{OE}$, $\overline{PPU}$ and $\overline{WKPU}$). The ports (3 / 4) have a PxREG and a PxPIN SFR register (see Figure 4 for absolute locations), but it does not have a PxSSEL or a PxIO SFR register. This means that the ports CANNOT be structured as PUSH/PULL or open drain. The circuit is hard wired to configure these pins as OPEN DRAIN only.

As stated in the Port 1/2 and 6 section, the $\overline{WKPU}$ signal gets turned off by the write to the PxSSEL register. These ports don't have a PxSSEL register, therefore the $\overline{WKPU}$ signal is connected directly to the $\overline{RESET}$ signal. This means that while the $\overline{RESET}$ is active (low) the $\overline{WKPU}$ is also active. The $\overline{WKPU}$ signal is turned off when the RESET# goes inactive.

The $\overline{PPU}$ signal that is usually connected to an R-C circuit and triggered by $\overline{RESET}$ FALLING edge is NOT connected. The $\overline{PPU}$ signal is ALWAYS inactive.

The last thing to note about PORT 3 and 4 hardware is the $\overline{OE}$ signal. Since there is no PxIO bit associated with each pin of these ports, the $\overline{OE}$ signal is always an open drain configuration. (On ports 1/2/5/6 the $\overline{OE}$ is: PxREG AND PxIO, on ports 3/4 the $\overline{OE}$ is: PxREG AND PxIO with PxIO hardwired to a "1" only - OPEN DRAIN.

Port 3 has an alternative function of Slave Port. Later revisions of this document will include discussions of this function.

It is also possible that future KX core devices will have push/pull capability on ports 3 and 4, when used as ports (not as system bus pins).

## 3.4 Port 5

The structure for PORT 5 is very similar to that of Ports 1 / 2 and 6. It too has Schmitt trigger CHMOS inputs and CHMOS Outputs (See previous sub-sections). However, some of the input circuitry has been optimized for high speed input capabilities.

When the port is configured as a system function, the P5IO register has NO affect on configuring the port.

For example, if the P5.0 pin (ALE/$\overline{ADV}$) is configured as a system function (ALE/$\overline{ADV}$), the port will be configured as Push/Pull regardless of the value in the P5IO register. This is true for all of port 5 pins. The system function defines whether the port is push/pull output or an input pin. (Open drain output is not a feature of the system functions).

The $\overline{RD}$ (P5.3) is also special in that the $Q_D$ and $Q_U$ transistors are stronger for higher loading capabilities.

$\overline{WKPU}$ and $\overline{PPU}$ signals are also present on port 5 and need to be dealt with accordingly by writing to P5SSEL) register to turn off the 150K Ohm reset pull-up.

After RESET the port 5 pins are configured as LSIO. If the $\overline{EA}$ is strapped to run from external memory, the ALE/$\overline{ADV}$ (P5.0) and the $\overline{RD}$ (P5.3) are configured as system functions.

Configuring these pins as system functions will allow the device to read the CCB0 (2018H) and CCB1 (201AH) from the external memory device.

The rest of port 5 will either be configured by the software program or by the CCB0/CCB1 reads.

For example: The READY (P5.6) pin comes out of reset as a LSIO *not* as READY. If the CCB0 and CCB1 data directs the 8XC196KR device to use the READY pin to control wait states *and* the IRC field is = "111" (max wait states are defined by ready), the device could lock up if the READY pin were to continue being an LSIO pin.

In this case the internal logic would re-configure the READY pin as a system (READY) function. Hence, no lock up condition.

If other pins on Port 5 are used as system functions, the PxSSEL register must be written. This will turn off the $\overline{WKPU}$ signal and configure the port as either LSIO or system function.

*Note that the SLPINT (P5.4) has a third function. If this pin is driven to a logical low on the rising edge of $\overline{RESET}$, the KR device will enter the ONCE (ON-Circuit Emulation) test mode.* This mode tri-states all device pins except power pins and XTAL1 / XTAL2.

For the above reason, it is recommended that the P5.4 (SLPINT) pin be used as an OUTPUT ONLY pin. The internal weak pullup will insure that this pin is high prior to $\overline{RESET}$ rising edge is not loaded.

| | PORT 5 Truth Table | | | | | |
|---|---|---|---|---|---|---|
| PxIO | 0 | 0 | 1 | 1 | X | X |
| PxREG | 0 | 1 | 0 | 1 | X | X |
| PxSFIO | X | X | X | X | 0 | 1 |
| PxSSEL | 0 | 0 | 0 | 0 | 1 | 1 |
| $Q_U$ | off | on | off | off | off | on |
| $Q_D$ | on | off | on | off | on | off |
| PxPIN | Low | High | Low | HZ* | Low | HZ |
| Port Config. | Push/Pull | | Open Drain | | Push/Pull | |
| Port Function | LSIO | | | | System | |

* During RESET and until first write to PxSSEL, $\overline{\text{WKPU}}$ is active.

**Figure 3-5. Port 5 Truth Table**

## 3.5 Top 5 Issues With the Ports

(1) Setup the ports to match the applications needs:

1. Write to PxREG
2. Write to PxIO
3. Write to PxSSEL

(2) P2.2/EXTINT, The external interrupt is a special port function. Therefore, EXTINT function is selected by setting PxSSEL bit 2.

(3) If the $\overline{\text{EA}}$ is strapped to run from external memory (low), the ALE and $\overline{\text{RD}}$ pins are hardwired to system functions.

(4) The 150K Ohms weak pullup resister is turned on by the $\overline{\text{RESET}}$ low signal, and turned off by the first write to the PxSSEL register.

(5) Beware of the SLPINT/P5.4 being driven low on the rising edge of $\overline{\text{RESET}}$ (this will enter ONCE mode).

## 4.0 SERIAL I/O PORT (SIO PORT)

The serial port on the 8XC196KR has one synchronous mode and three asynchronous modes. Both the receiver and the transmitter are double buffered. This allows for the reception of a second byte before the first byte has been read, and uninterrupted transmissions, respectively. The synchronous mode (MODE 0) is often used for shift register based I/O expansion. Mode 1 is an 8 bit asynchronous mode used for normal communications like RS-232C, while modes 2 and 3 are 9 bit data asynchronous modes which are specially designed for multiprocessor communications.

The serial port on the 8XC196KR is capable of sending two distinct interrupts to the core; a receive (RI), and a transmit (TI) interrupt. There are separate mask bits for these two sources in the INT_MASK1 register which is located at 13H. These mask bits can be used to disable ("0") and enable ("1") interrupts to the core, but the RI and TI bits of the SP_STATUS register will be set regardless of the setting of the mask bits. There are separate RI and TI vectors which are located at 2038h and 2036h respectively.

## 4.1 Serial Port SFRs

Control and status of the serial port is accomplished by using five dedicated registers: the Serial Port Control register (SP_CONTROL) at location 1FBBh, the Serial Port Status register (SP_STATUS) at 1FB9h, the Serial Port Baud Rate Register (SP_BAUD) at location 1FBCh, the Serial Port Transmit Buffer Register (SBUF_TX) at location 1FBAh, and the Serial Port Receive Buffer Register (SBUF_RX) at location 1FB8h. These are all byte addressable registers except SP_BAUD. It is word addressable. A map of these registers is shown below in Figures 4- 1, 4-2, and 4-3.

**SP_CONTROL** 1FBBH:byte

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| X | X | X | TB8 | REN | PEN | M2 | M1 |

**TB8** 9th Bit for transmission

**REN** Enables the receiver

**PEN** Enables Parity (even)

**M2,M1-** 00 Mode 0 / Sync
01 Mode 1 / Async (std)
10 Mode 2 / Async (9th bit enable)
11 Mode 3 / Async (9th bit data)

**Figure 4-1. SP_CONTROL Register**

#### 4.1.1 SP__CONTROL

The SP__CONTROL register controls various aspects of the serial port's operation. The lower two bits (bit 1 and bit 0) selected which mode the serial port is in: Mode 0 = 00, Mode 1 = 01, Mode 2 = 10 and Mode 3 = 11.

Bit 2 of SP__CONTROL controls whether parity is being used. When it is high (1), even parity for mode 1 or mode 3 is selected. NOTE: Parity cannot be enabled for mode 2.

Bit 3 is used to enable the receiver (RXD to SBUF__RX).

Bit 4 is used to determine the setting of the ninth data bit in modes 2 and 3 during transmissions. It is cleared after each transmission and must be reset for each data byte whose 9th bit is to be set.

Bits 7, 6, and 5 of the SP__CONTROL register is reserved and should be written as zeros for compatibility with future products.

#### 4.1.2 SP__STATUS

The SP__STATUS register contains status information about the serial port. It is very important to keep in mind that when this register is read, the RPE, TI, RI, OE, and FE bits will be cleared. This mandates the use of a shadow register (see example program 3) if more than one bit is to be tested.

**SP__STATUS** 1FB9H:byte

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| RB8/RPE | RI | TI | FE | TXE | OE | X | X |

| | |
|---|---|
| **RB8** | Set if 9th bit set (no parity) |
| **RPE** | Set if parity enabled and parity error |
| **RI** | Set after last data bit received |
| **TI** | Set at End of last data bit sent |
| **FE** | Set if no STOP bit found |
| **TXE** | Set when byte is in SBUF__TX |
| **OE** | Set if overrun error occurred |

**Figure 4-2. SP__STATUS Register**

Bit 2 is the Overflow Error bit, and is set when a new byte is loaded into SBUF__RX (by the receiver) before the previous byte has been read. This alerts the user that some information was lost because it wasn't read in time. In this case, SBUF__RX will contain the last byte received. The old data is lost.

The Transmitter Empty (TXE) bit (bit 3) is set if the transmit buffer is empty and SBUF__TX register is able to take up to two bytes. TXE will be cleared as soon as the first byte is written to SBUF__TX register. Only one byte may be written to SBUF__TX if TI alone is set. The TI bit (bit 5) is set as soon as transmission of the last data bit is complete and so indicates that the transmitter is ready to take another byte.

Bit 4 is the framing error bit. It gets set when a valid stop bit can not be found for a received byte.

*Note: The 8XC196KR A-step device will not generate a framing error if the last data bit sent is a 1. The part needs to see a low to high transition in order to detect the stop bit. This is fixed on later steppings.*

The RI bit (bit 6) is set after the last data bit is sampled. This happens approximately in the middle of the bit time. It should be noted that if the SIO port is run in loop-back mode (with the transmitter and receiver tied together), the transmit flag (TI) will be set approximately 1 bit time before the receive interrupt (RI) flag is set.

The Receive Bit 8 (RB8, bit 7) is used when the port is configured in modes 2 or 3. This bit is set when the 8th data bit is set (counting from 0). The other function of this bit is the receive parity error bit (RPE). This will be set if an even parity error was detected by the receiver.

Bits 1 and 0 of the SP__STATUS register are reserved. Any data in these bits is to be ignored.

### 4.2 Baud Rate Generation

The SIO has a dedicated Baud Rate Generator clock. Baud rates for all modes are determined by the contents of a word register (SP__BAUD) at 1FBCh. While this is a 16 bit register, only the lower 15 bits actually determine the baud rate. The MSB selects one of two sources for the input frequency to the baud rate generator. When it is set to a 1, the frequency on the XTAL1 pin is selected as a source to the baud rate clock. If it is 0, the frequency on the T1CLK pin (P6.2) is used. SP__BAUD is a write only register; it will read all 1's. It can be updated via word writes to locations 1FBCh.

The value to be placed in SP__BAUD for a given baud rate depends on both the mode and clock selected and can be calculated as follows:

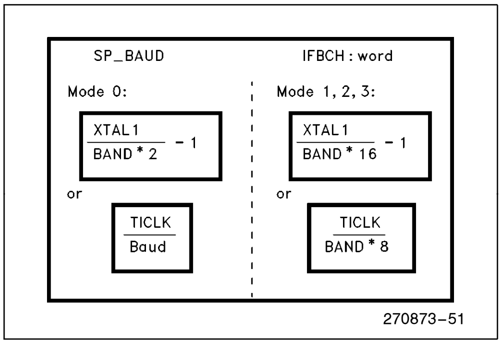



**Figure 4-3. SP__Baud Register Equations**

The equations used are different for both the synchronous and asynchronous modes, and for the internal (XTAL1) or external (T1CLK) clocks. The maximum frequency on the T1CLK pin is 4MHz. The T1CLK input is NOT prescaled.

The following table lists various baud rates, the value to be programmed into SP__BAUD, and the error associated with the resulting baud rate.

| Baud Rate | SP__BAUD Value Mode 0 | SP__BAUD Value Others | % Errors |
|---|---|---|---|
| 9600 | 8340H | 8067H | 0.16% |
| 4800 | 8682H | 80CFH | 0.16% |
| 2400 | 8D04H | 81A0H | 0.16% |
| 1200 | 9A0AH | 8340H | 0.04% |
| 300 | E82BH | 8D04H | 0.01% |

**Figure 4-4. Common Baud Rate Values**

## 4.3 SIO Port Configuration

Before the SIO unit can be used, two port 2 I/O pins (P2.0 and P2.1) MUST be configured. This is handled by writing to the P2SSEL register (1FC9h), the P2IO register (1FCBh), and the P2REG (1FCDh). Note that these are byte registers (See port section for details).

Setting the P2SSEL.0 bit to a 1 tells the port logic that pin P2.0 is to be controlled using an internal special function source and not act as a general purpose I/O port pin. Clearing P2IO.0 causes pin 0 to become a push/pull output. Similarly, if the RXD pin is to be used as an input/output, bit 1 of P2SSEL, P2IO, and P2REG must all be set to 1. Writing a 1 to P2IO.1 configures pin P2.1 to become an input/open drain output. Forcing a 1 in P2REG.1 is needed to insure that the pull down associated with that pin is turned off. Thus, RXD may be used, in mode 0, as both an input and an output with an external pullup The following code segment demonstrates how to set up port 2 for use with SIO.

```
; ***********************************************
; *    PORT 2 SFR'S used with windows
; ***********************************************
P2IO_W      EQU 0EBH:BYTE   ; Window to 1FCBH
P2SSel_W    EQU 0E9H:BYTE   ; Window to 1FC9H
P2REG_W     EQU 0EDH:BYTE   ; Window to 1FCDH

;Port_Configure
    LdB  WSR, #7Eh          ; Window 1fc0-1fdf to e0-ff
    OrB  P2REG_W, #02h      ; Force 1 at P2.1
    OrB  P2IO_W, #02h       ; set P2.1 as IN/OD output
    AndB P2IO_W, #0FEh      ; set P2.0 and P/P output
    OrB  P2SSel_W, #03h     ; Set P2.0 and P2.1 for SIO
```

270873–20

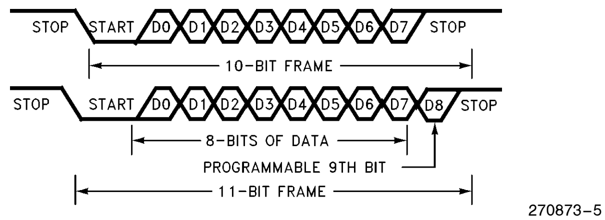



**Figure 4-5. Serial Port Frames, Mode 1, 2 and 3**

## 4.4 Mode 0: Synchronous Communications

Mode 0 is a synchronous mode which is commonly used for shift register based I/O expansion. In this mode the TXD pin outputs the clock (a set of 8 pulses) and the RXD pin either transmits or receives the data. Data is transmitted/received 8 bits at a time with the LSB first.

Mode 0 can be entered by first setting up port 2 as described above, and then setting up the SIO by writing the desired baud rate to the SP__BAUD register and writing the proper control value to the SP__CONTROL register. P2.1 must have an external pullup attached as it is configured as an input/open-drain output.

Reception starts when a 1 is written to the REN (Receiver Enable) bit in the serial port control (SP__CONTROL) register. If REN is already high, clearing the RI flag will start a reception. After the reception is complete, the RI flag will be set and an RI interrupt will be generated if enabled. In order to avoid a partial unwanted reception, the receiver must be disabled by clearing the REN bit in the SP__CONTROL register before the RI bit is cleared. If the SP__STATUS register is read before writing to SP__CONTROL, RI will automatically be cleared, thus starting a new reception.

Starting a transmission in mode 0 requires writing a byte to SBUF__RX register. A set of 8 pulses will be sent out from the TXD pin, and the data will be sent out of the RXD pin. NOTE: This is the only mode for which RXD can be used as an output. After the data has been shifted out, TI will be set and a TI interrupt will be generated if enabled.

## 4.5 Mode 1: Standard Asynchronous Serial I/O

Mode 1 is a standard asynchronous communications mode. The data frame used is shown in Figure 4-5. It consists of 10 bits; a start bit, 8 data bits, and a stop bit. If parity is enabled (PEN = 1) then a parity bit is sent in place of the last data bit. Only even parity is supported on the 8XC196KR. Parity is also checked upon reception, and if an error is detected, RPE (Receiver Parity Error) in SP__STATUS is set.

The transmit and receive functions are controlled by separate clocks, but both clocks operate at the same frequency. The transmit clock starts as soon as the baud rate generator is initialized, but the receive shift clock is reset when a 1 to 0 transition is received (signaling a start bit reception).

```
;  ************************************
;  *    SIO SFRs used with windows
;  ************************************
SP_BAUD_W      EQU    0FCH:WORD          ; Window to 1FBCH
SP_STATUS_W    EQU    0F9H:BYTE          ; Window to 1FB9H
SP_CONTROL_W   EQU    0FBH:BYTE          ; Window to 1FBBH
SBUF_RX_W      EQU    0F8H:BYTE          ; Window to 1FB8H
SBUF_TX_W      EQU    0FAH:BYTE          ; Window to 1FBAH
;  ***********************************************
;  *    PORT 2 SFR'S used with windows
;  ***********************************************
P2IO_W       EQU 0EBH:BYTE               ; Window to 1FCBH
P2SSel_W     EQU 0E9H:BYTE               ; Window to 1FC9H
P2REG_W      EQU 0EDH:BYTE               ; Window to 1FCDH
;  *********************
;  *  program equates  *
;  *********************
shadow_stat  EQU 0C0h:BYTE               ; shadow register for SP_STATUS
buffer       EQU 0C1h:BYTE               ; temp storage for I/O
;  ******************
;  *  main routine  *
;  ******************
        cseg at 2080h
Start:
        DI                               ; Disable Interrupts
        DPTS                             ; Disable PTS
        LD   SP,#0500h                   ; Initialize Stack
        LDB  shadow_stat,#00h            ; clear shadow register
        LDB  WSR,#07Eh                   ; Window 1fc0-1dff to e0-ff
;Port_Configure
        ORB  P2REG_W, #02h               ; Force 1 at P2.1
        ORB  P2SSel_W,#00000011b         ; Set P2.0 and P2.1 for SIO
        ORB  P2IO_W,#00000010b           ; set P2.1 for input (RXD)
        AndBP2IO_W,#11111110b            ; set P2.0 for output (TXD)
        LDB  WSR,#07Dh                   ; Window 1fa0-1fbff to e0-ff
;SIO_Configure
        LD   SP_BAUD_W,#8067h            ; Set to 9600 baud, XTAL1=16Mhz
        LDB  SP_Control_W,#00001001b     ; Mode 1, receiver enb, no parity
begin_IO:
        LDB  SBUF_TX_W,#0EAh             ; send data out
loop:
        ORB  shadow_stat,SP_STATUS_W     ; read status of port
        JBC  shadow_stat,6,Check_trans   ; if RI=0, check TI
        LDB  buffer,SBUF_RX_W            ; else read input buffer
        AndBshadow_stat,#0BFh            ; clear shadow RI bit
Check_trans:
        JBC  shadow_stat,5,loop          ; if TI=0, loop back
        LDB  SBUF_TX_W,buffer            ; else write out data
        AndBshadow_stat,#0DFh            ; clear shadow TI bit
        SJMP loop                        ; and so on
END
```

270873–21

**Program 3. SIO Communication via Polling the SP__STATUS Bits (TI and RI)**

### 4.5.1 SETTING UP MODE 1 OPERATION:

The following code demonstrates how to set up mode 1 with no parity at 9600 baud, assuming a 16MHz. Also, polling the RI and TI status bits with the use of a shadow register is demonstrated. It is very important to set up port 2 properly since the serial port shares pins with this port.

The next section of code sets up the serial port. First, the baud rate register, SP__BAUD is programmed. The high bit is set, indicating that the internal clock, XTAL1, is to be used. The control register, SP__CONTROL, is set for mode 1, no parity, and the receiver is enabled.

Reading the SP__STATUS register will automatically clear various bits, so a shadow register is used because the contents are to be checked twice (once for RI and once for TI bits) so that they can be acted upon later. The above program illustrates several points about using a shadow register for the SP__STATUS register. First, the shadow register needs to be updated to reflect the current status of the SP__STATUS register. This is handled by ORing the shadow__stat register with the SP__STATUS register. Second, it is important to clear the various flags in the shadow__stat register after they have been acted upon. This is handled by clearing the RI bit in the shadow register after reading the data in the buffer, and by clearing the TI bit in the shadow register after transmitting new data.

```
MCS-96 MACRO ASSEMBLER    PTS_SIO                                    02/01/90 02:05:41 PAGE 1

DOS 3.30 (038-N) MCS-96 MACRO ASSEMBLER, V1.1

SOURCE FILE: PTS_SIO.A96
OBJECT FILE: PTS_SIO.OBJ

ERR LOC  OBJECT          LINE                       SOURCE STATEMENT
                1       $debug
                2       $nolist
              287       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              288       ;
              289       ;
              290       ; This program demonstrates the use of the serial port in
              291       ; in mode 1.  The PTS is also used to both send and receive
              292       ; data.
              293       ;
              294       ; This program was tested on the '196kr SBE with the serial
              295       ; ports receiver and transmitter tied together.  Place a
              296       ; jumper from E5-A to E6-A if you wish to run this code on
              297       ; the SBE.
              298       ;
              299       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              300       ;;;
              301       ;;;   windows for SRFs
              302       ;;;        1f
              303       ;;;
00C9          304       P2SSEL_W                  EQU    0C9h:byte     ; window to 1fc9
00CD          305       P2REG_W                   EQU    0CDh:byte     ; window to 1fcd
00CB          306       P2IO_W                    EQU    0CBh:byte     ; window to 1fcb
00BC          307       SP_BAUD_W                 EQU    0BCh:word     ; window to 1fbc
00BB          308       SP_CONTROL_W              EQU    0BBh:byte     ; window to 1fbb
00B9          309       SP_STATUS_W               EQU    0B9h:byte     ; window to 1fb9
00BA          310       SBUF_TX_W                 EQU    0BAh:byte     ; window to 1fba
00B8          311       SBUF_RX_W                 EQU    0B8h:byte     ; window to 1fb8
              312
              313
2036          314          CSEG    AT      2036H          ;sets up pointer to TI interrupt service
2036    E920   315      dcwTI_ISR
              316
2038          317          CSEG    AT      2038h          ;sets up pointer to RI interrupt service
2038    F720  318       dcwRI_ISR
              319
2056          320          CSEG    AT      2056h          ;pointer to PTS control block for TI
2056    E000  321       dcwTI_PTS_CNT
              322
2058          323          CSEG    AT 2058h               ;pointer to PTS control block for RI
2058    F000  324       dcwRI_PTS_CNT
              325
        00F8  326          RSEG    AT      0F8H           ;PTS transmit control Block
        00F8  327 TI_PTS_CNT:   DSB     1                 ; Transmit Interrupt Count register
        00F9  328 TI_PTS_CON:   DSB     1                 ; Control register
        00FA  329 TI_PTS_SRC:   DSW     1                 ; Source pointer
        00FC  330 TI_PTS_DST:   DSW     1                 ; Destination pointer
              331
```

270873–22

**Program 4a. Using the PTS with both the TI and RI Interrupts**

### 4.5.2 SIO AND THE PTS

The final example in this section demonstrates the use of interrupts and the PTS with the serial port.

In this example both the RI and TI interrupts are being used with the PTS. Each PTS control block is programmed for the single transfer mode.

The RXD and TXD pins are strapped in loop back mode and the message being sent out the TXD is being received in the RXD and placed into external RAM at location 221Dh through 231Ch.

The serial port is configured for 9600 baud using a external XTAL of 16 MHz. The Port 2 pins are configured accordingly (TXD = push/pull, RXD = open drain-input, and both are special function selected, P2SSEL = 3h).

```
00F0                     RSEG    AT     0F0h     ;PTS receive control Block
00F0                 RI_PTS_CNT: DSB 1           ; receive interrupt count register
00F1                 RI_PTS_CON: DSB 1           ; control register
00F2                 RI_PTS_SRC: DSW 1           ; Source pointer
00F4                 RI_PTS_DST: DSW 1           ; Destination pointer

2080                     CSEG    AT     2080H
2080             TI_START:
                 ;
                 ;  Set up port 2 and serial port control
                 ;
2080  B11F14     LdB  WSR, #1FH                  ; Open window 1FH
2083  A1004018   Ld   SP,#0500h                  ; init stack CODE RAM
2087  B10DBB     ldb  SP_CONTROL_W, #0DH         ; mode 1 enable rec and parity
208A  A16780BC   ld   SP_BAUD_W, #8067H          ; baud rate = 9600 at 16MHz
208E  9102CD     orb  P2REG_W, #02h
2091  9102CB     orb  P2IO_W, #02H               ;  RxD = Input
2094  71FECB     andb P2IO_W, #0FEh              ;  TxD = Output
2097  9103C9     orb  P2SSEL_W, #03H             ; Select TxD and RxD peripheral
209A  1114       ClrB WSR
                 ;
                 ;set up PTS transmit control block
                 ;
209C  B1FEE0     Ldb  TI_PTS_CNT, #254           ; load count with 254 characters
209F  B19AE1     Ldb  TI_PTS_CON, #10011010B     ; single byte transfer mode
20A2  A10521E2   Ld   TI_PTS_SRC, #TABLE         ; point to the beginning of the table
20A6  A1BA1FE4   Ld   TI_PTS_DST, #SBUF_TX       ; point to SBUF_TX as destination
                 ;
                 ;set up PTS receive control block
                 ;
20AA  B1FFF0     Ldb  RI_PTS_CNT, #255           ;255 since entire recp. done by PTS
20AD  B195F1     Ldb  RI_PTS_CON, #10010101B
20B0  A11D22F4   Ld   RI_PTS_DST, #BUFFER        ; point to destination buffer
20B4  A1B81FF2   Ld   RI_PTS_SRC, #SBUF_RX       ; point to SBUF_RX as source

20B8  B11813     LdB  INT_MASK1, #00011000B      ; Enable RI and TI interrupts
20BB  A1001804   Ld   PTS_SELECT, #1800H         ; Enable RI and TI of PTS
20BF  B1FFD0     Ldb  0D0h,#0FFh                 ; flag for PTS routines complete
20C2  B2E3D1     LdB  0D1h, [TI_PTS_SRC]+
20C5  C701BA1FD1 StB  0D1h, SBUF_TX              ; start transmission
                 EPTS                            ; enable PTS
20CB  FB         EI                              ; enable interrupt
                  ;
                  ; The PTS will now handle the transmission and reception of characters.
                  ; When a PTS routine is done, the  TI (or RI) interrupt routine below
                  ;   will be called.  These routines just clear a flag to note that the
                  ;   PTS routine is finished.
20CC             wait:
20CC  9900D0           cmpb          0D0h,#00h
20CF  D7FB             Jne           wait                ;wait around for PTS to finish
                                                         ;useful stuff done here
                       dpts                              ;turn off the PTS
20D2  B17D14           Ldb           WSR,#7Dh            ;activate window 7Dh
20D5  B101BB           Ldb           SP_CONTROL_W,#01h   ;disable receiver
20D8  90B97F           OrB           07Fh,SP_STATUS_W    ;Read status of port
20DB  3F7F07           JbS           07Fh,7,P_error      ;check for parity error
20DE  3C7F06           JbS           07Fh,4,Fr_error     ;check for framing errors
20E1  1114             Clrb          WSR
20E3  27FE             Br            $

20E5              P_error:
20E5  27FE             Br            $

20E7              Fr_error:
20E7  27FE             Br            $
```

270873–23

**Program 4b. Using the PTS with both the TI and RI Interrupts**

```
                         ; There is a bug with the interrupt server on the 196.  It is possible for
                         ;the wrong routine to be called.  During the latency period for a normal
                         ;interrupt, if a PTS interrupt occurs, the normal interrupt routine for the
                         ;PTS interrupt will be erroneously executed.
                         ;
                         ; Due to this bug, it is possible to enter these interrupt service
                         ;routines before the PTS transfer is complete.  Therefore it is necessary
                         ;to check to see if the PTS_SELECT bit for a given routine is set.  If
                         ;it is, a PTS call should have been made, so the routine aborts and
                         ;forces the call be setting the proper INT_PEND bit.

20E9                     TI_ISR:

                                          PUSHA
20EA  3B0505                              JBS    PTS_SELECT+1,3,abort1
20ED  710FD0                              AndB   0D0h,#0Fh          ; mark Xmittion complete
                                          POPA
20F1  F0                                  RET
20F2                     abort1:
20F2  91081                               ORB    INT_PEND1,#08h  ; PTS to correct bug
                                          POPA
20F6  F0                                  RET

20F7                     RI_ISR:

                                          PUSHA
20F8  3C0505                              JBS    PTS_SELECT+1,4,abort2
20FB  71F0D0                              andb   0D0h,#0F0h         ; mark RI complete
                                          POPA
20FF  F0                                  RET
2100                     abort2:
2100  911012                              ORB    INT_PEND1,#10h  ; force PTS RI routine
                                          POPA
2104  F0                                  RET

2105                     Table:
2105  5468697320697320   DCB
                         'This is a test of the PTS and the Serial Port',0AH,0DH ;47
2134  4920616D20707269   DCB
                         'I am printing out 255 characters via the SIO ',0AH,0DH ;94
2163  6174203936303020   DCB
                         'at 9600 baud, 16MHz operating frequency, and ',0AH,0DH ;141
2192  6D6F64652031206F   DCB
                         'mode 1 operation.  1234567890123456790123456 ',0AH,0DH ;188
21C1  3738393031323334   DCB
                         '78901234567890123456789012345678901234567',0AH,0DH ;231
21EC  636F64652062792E   DCB
                         'code by......SMc/RMK',0AH,0DH,0AH,0D              ;255
2204                     Table_end:
2204  2020202020202020   DCB                      '                         ' ; Buffer ending
221D                     BUFFER:                  ; start of PTS RX input buffer

221D                                      end
```

270873–24

**Program 4c. Using the PTS with the TI and RI Interrupts**

The PTS control blocks (both RI and TI) are initialized and the interrupt mask and PTS__SELECT bits are also set.

Lastly the program sends the first byte from the buffer to the SIO SBUF Transmit register. This starts the transmission/reception process rolling.

The TI PTS cycle send the data out the SBUF__TX. The RI PTS cycle receives the byte and transfers it to external RAM locations 221D to 231Ch.

After the PTS transfers are completed a normal software interrupt request (for both RI and TI) is executed. This will flag to the main line program that the PTS is completed.

Lastly the receiver (RXD) is disabled, and the Parity and Framing errors are checked.

## 4.6 Modes 2 and 3: 9 Bit Communications Modes

Modes 2 and 3 are asynchronous 9 bit communications modes. In mode 2, parity can NOT be enabled. However, the 9th bit is used to determine whether or not a receive interrupt will occur. If the 9th bit being received is set, RI will be set and a receive interrupt will occur. This allows for a selectable reception link.

For transmission in mode 2, the state of the 9th bit is determined by the setting of bit 4 (TB8). If TB8 is 1, the 9th bit will be set. TB8 is cleared after each transmission and so it must be written before each ("1") transmission.

Mode 3 is similar to mode 2 except that parity can be enabled and that the receiver will generate an interrupt (set RI) every time a byte is received (independent of the state of the 9th bit).

Modes 2 and 3 work very well together in a multi-processor environment. The master processor will operate in mode 3. While the slaves will usually operate in mode 2. When the master wants to talk to a slave, it will first set the 9th bit high with the address byte. The slaves (operating in mode 2) will be interrupted and the one that is being addressed can switch to mode 3. The two processors can then talk (with the 9th bit clear), and the other slave processors will not be interrupted.

Setting up modes 2 and 3 is just like setting up the other modes. Port 2 must be set up, the baud rate written, and the control register programmed.

Below are two program segments (one for the master and one for a slave) which demonstrate the use of these modes.

NOTE: THE SEGMENTS ILLUSTRATED ARE JUST THAT. THEY ASSUME THAT THE PORTS ARE SET UP AS NEEDED.

```
; MASTER
        CSEG
Initial_Start:
    DI                          ; Disable Interrupts
    DPTS
    LD      SP,#0500h           ; Initialize Stack
    LdB     Wsr,#7Eh            ; 1fC0-1fdf -> e0-ff
    Clr     Time1

SIO_Configure:
    LdB     Wsr,#07Dh             ; 1fa0-1fbf -> e0-ff
    LD      sp_baud_W,#8067h    ; Enable 9600 baud
    LDB     sp_control_W,#1Bh   ; Mode 3, Rec En, TB8
cal:
    LDB     SBUF_TX_W,#03h      ; See if slave #3 is there
cntl:
    DJNZ    Time1,cntl
    DJNZ    Time1+1,cntl          ; Wait about .07 second
    JBC     sp_status_w,6,nores; slave respond
    CmpB    0f8h,#03h             ; correct slave?
    JNE     error
    LDB     slvstat,#0ffH         ; slave respond flag
;
; do whatever is needed to complete transaction with
;    slave
;
nores:
    CLRB    WSR     ; No windows open
Self:
    SJMP    Self
error:
    SJMP    error
```

```
; SLAVE

    cseg at 2038    ; set up RI interrupt pointer
    DCW     RI_ISR

    CSEG

Initial_Start:
    DI          ; Disable Interrupts
    DPTS
    LDB  mode,#02h     ; we start in mode 2
    LD   SP,#0500h     ; Initialize Stack
    LDB  WSR,#7Eh      ; Window 1fC0-1fdf -> e0-ff

SIO_Configure:
    LDB  WSR,#07Dh              ; 1fa0-1fbf -> e0-ff
    LD   SP_BAUD_W,#8067H       ; Enable 9600 baud
    LDB  SP_Control_W,#0Ah      ; Mode 2, Rec Enabled
    ORB  INT_MASK1,#10h         ; Enable Rec Int
    EI                          ; enable interrupts
self:       SJMP    self
;
; RI_Interrupt Service Routine
;
RI_ISR:
    PUSHA
    INCB    80h             ; count times in interrupt
    CMPB    mode,#03h       ; see if we are in mode3
    JE      cont            ; if not, check ID
    CMPB    0f8h,#my_id     ; is master talking to me?
    JNE     out             ; no so just leave
    LDB     0fbh,#0Bh       ; yes switch to mode 3
    LDB     mode,#03h       ; note mode switched
    ldb     0fah,#03h       ; echo slave id number
    INCB    81h
```

270873–25

```
out:
    POPA
    RET

cont:
; do whatever is needed here
; like interpret commands, transfer data, ect.
;
    POPA
    RET
END
```

270873-26

There are a few points which should be emphasized about the above program segments. First, the master always operates in mode 3. Second, the slave operates in mode 2 while it is idle, but switches to mode 3 when the master is talking to it. Finally, the master only sets the 9th bit when it is sending out an address of the slave that it wishes to communicate with. This is done so that the other slaves will not be interrupted unless an address is being broadcast over the serial link.

## 4.7 Top 5 Issues with the SIO

(1) When using the T1CLK as the clock to the serial port, make sure that the pin (P6.2) is special selected (P6SSEL bit 2 = 1).

(2) When using the RI and TI flags in the SP__STATUS register, use a shadow register in RAM.

(3) Make sure that the SIO port pins are set up according to the application:

1. Write to P2REG
2. Write to P2IO
3. Write to P2SSEL

(4) Parity is not possible with Mode 2.

(5) SBUF__TX and SBUF__RX registers are separate register (unlike past 8096/80C196 devices) and are BOTH double buffered.

# 5.0 SYNCHRONOUS SERIAL I/O AND PERIPHERAL TRANSACTION SERVER

The Synchronous Serial I/O (SSIO) unit of the 8XC196KR has two identical channels, each capable of transmit and/or receive functions, with a shared baud rate generator. The SSIO will provide simultaneous bi-directional communications between synchronous serial I/O devices (as between two KR processors, or other serial peripherals). Listed below are a few of the many modes possible with the SSIO:

- **Master Transceiver** (half duplex mode, single SSIO channel)
- **Slave Transceiver** (half duplex mode, single SSIO channel)
- **Dual Channel Masters with common clock** (full duplex, lock-step synchronous)
- **Dual Channel Slaves with common clock** (full duplex, lock-step synchronous)
- **Dual Channel with different clocks** (master transmits clock, slave receives clock)

The two channels share a single baud bate generator. It internally provides baud rates from 15.75K baud to 2 Meg at 16MHz. Both channels can be used with the Peripheral Transaction Server (PTS) using the Handshake mode.

Both SSIO channels have distinct interrupts to the core (XFR0 and XFR1). Each channel has a separate mask bit in the INT__MASK1 register located at 13H. The mask bits are used to enable ("1") or disable ("0") interrupts to the core. However, the INT__PEND1 XFR0 and XFR1 bits, will be set regardless of the setting of the mask bits.

SSIO channel 0 interrupt vector through location 2032H and channel 1 through location 2034H.

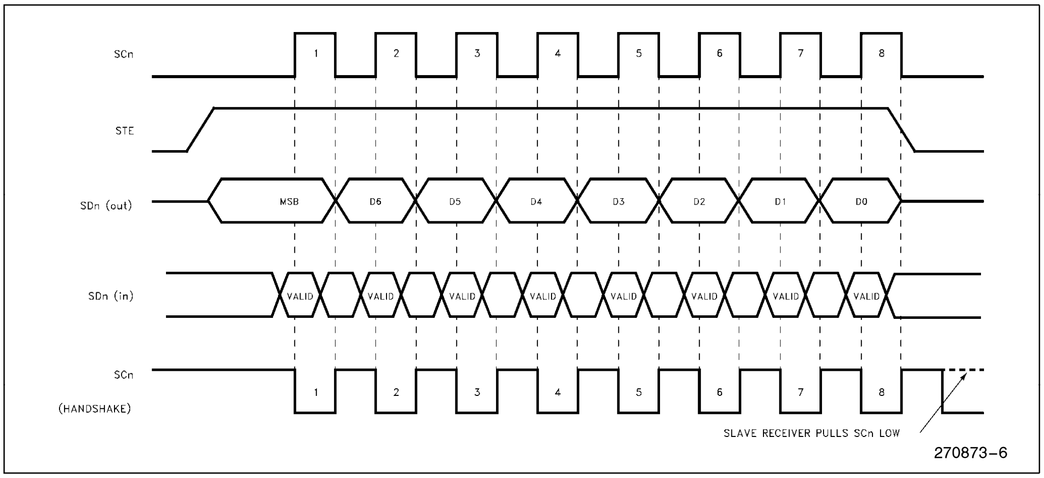



**Figure 5-2. SSIO Transmit/Receive Timings**

## 5.1 SSIO Port SFRs

Control of the Baud Rate Generator is accomplished by using the SSIO__BAUD Register at location 1FB4H. This register includes an enable bit (bit 7, 1 = enable) and seven bits to select a baud rate (bits 6 through 0) from 15.75 KHz to 2 MHz using a 16 MHz crystal or 11.82 KHz to 1.5 MHz with a 12 MHz crystal. After reset this register is cleared, resulting in the generator being disabled with a clock value of 2 MHz (16 MHz crystal). Each SSIO channel can be clocked from an external source, through the SC0 or SC1 pins (slave mode).

While writes to the SSIO__BAUD register will enable and set the baud rate value, reads of this register will return the current state of the baud rate clock with bit 7 returning the clock pin current state and bits 0 to 6 returning the current down count in the generator.

Control registers SSIO__STCR0 or SSIO__STCR1 are used to configure the two SSIO channels. The following figure defines the functions for each bit:

**SSIO__STCRn** byte 1FB1h and 1FB3h

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| M/$\overline{S}$ | T/$\overline{R}$ | TRT | THS | STE | ATR | OUF | TBS |

**M/$\overline{\textbf{S}}$** Master/$\overline{\text{Slave}}$
**T/$\overline{\textbf{R}}$** Transmit/$\overline{\text{Receive}}$
**TRT** Transmitter/Receiver Toggle
**THS** Transceiver Handshake Select
**STE** Single Transfer Enable
**ATR** Auto Transfer Re-enable
**OUF** Overflow/Underflow Flag
**TBS** Transceiver Buffer Status

**Figure 5-1. SSIO Control Register**

Bit 7 defines the mode of the channel. A "1" will program MASTER, while a "0" will program SLAVE. This refers to whether the SSIO clock pin will either transmit (master) or receive (slave) a clock.

*At this point it is important to note that the SC0 and SC1 pin is fed back internally to clock the channel. Even when in MASTER mode the clock is fed back. Because of this, the SCx pin MUST be configured as Special Function pins (P6SSEL bits 4 and 6 = 1). It is not possible to shift the data out without the SCx pin selected as special function.*

Bit 6 controls whether the channel is to receive ("0") or transmit ("1") data in/out of the data pin (SD0, SD1).

Bit 5 controls whether bit 6 is to be toggled to the opposite state ("1") after transmission or reception is complete, or left alone ("0"). Setting bit 5 will change the channel from an output to an input or vice versa, after each byte reception or transmission.

Bit 4 controls whether the channel is to be used as handshake ("1") or as a normal shift register ("0"). See handshake mode for details.

Bit 3 controls transmission or reception enabling. If this bit is set ("1") it will either transmit the data when the SSIO__STBx register is loaded, or enable reception of data in the SSIO__STBx register.

Bit 2 controls whether bit 3 is to automatically re-enabled (set) when reception or transmission is complete. With this feature, the user need only write to the SSIO__STBx register and the SSIO peripheral will transmit immediately.

The last two bits of this register are status bits. Bit 1 refers to data integrity (overflow/underflow). This bit is initialized by the user program, and updated by the core with each byte received or transmitted.

Bit 0 refers to the status of the SSIO__STBx register. The user code *MUST* initialize this bit. If the channel is a transmit channel, this bit set (1) means that the SSIO__STBx register is empty and waiting for data. If the channel is a receive channel, this bit cleared (0), means that the SSIO__STBx register is empty.

*Note that on RESET this register is cleared, setting the channel into a receive, slave mode, and the status of the SSIO__STBx is empty. If modified to a transmit, channel, make sure the user code sets (1) bit 0, indicating that the buffer (SSIO__STBx) is empty.*

## 5.2 Example 1

In this first example, the SSIO channel 0 sends data out and that data is received on the SSIO channel 1 (loop back mode). A Baud rate of 2 MHz is selected.

Both channels are jumpered together (clocks and data pins). It is important to first setup the port 6 pins prior to any SSIO transmissions or receptions.

Since Channel 0 is sending data in a master mode, SC0 (P6.4) and SD0 (P6.5) are set to be push/pull output pins. Conversely, Channel 1 is receiving both a clock and data and sets SC1 (P6.6) and SD1 (P6.7) as open drain input pins.

```
;     ****************************************************
;           KR SSIO Master Transmit/Slave Receive
;     ****************************************************
;
WSR       EQU    014h:Byte        ;
P6REG     EQU    0D5h:Byte        ; window to 1fd5h
P6IO      EQU    0D3h:Btye        ; window to 1fd3h
P6SSEL    EQU    0D1h:Byte        ; window to 1fd1h
BAUD      EQU    0B4H:Byte        ; window to 1fb4h
SSCR_0    EQU    0B1H:Byte        ; window to 1fb1h
SSCR_1    EQU    0B3H:Byte        ; window to 1fb3h
Ssio_Stb0 EQU    0B0H:Byte        ; window to 1fb0h
Ssio_Stb1 EQU    0B2H:Byte        ; window to 1fb2h
RESULT    EQU    122H:Byte        ;
;
; Send Data byte from SSIO0 to SSIO1 - > "RESULT"
    CSEG          AT       2080H  ;
    Ldb WSR, #1FH                 ; window 1fh
    Ldb P6REG, #0C0H              ; clr CH1, set CH0
```

270873–27

```
    Ldb P6IO, #0C0H               ; p/p CH1, in CH0
    Ldb P6SSEL, #0F0H             ; special function
    Ldb BAUD, #80H                ; 2 Meg Baud
    Ldb SSCR_0, #0C9H             ; master_xmit
    Ldb SSCR_1, #08H              ; slave_receive
    Ldb SSIO_STB0, #55H           ; xmit data 55
D_WAIT:
    JbC SSCR_1,0,D_WAIT           ; wait for received
    Stb SSIO_STB1, RESULT         ; store rec data

    Simp        $
```

270873–28

**Program 5. SSIO, Send One Byte**

In the above example, the port pins are configured, and the SSIO channel control registers are set to values that would either transmit or receive information in the SSIO__STBx registers.

Since only one byte is being received, only the STE bit in the control registers were set once. The write to the SSIO__STB0 register will send that byte out the SSIO channel 0 to channel 1's SSIO__STB1 register (since the two channels are in loop back mode).

## 5.3 Using the PTS and Handshake Mode

The SSIO Handshake Mode requires the output of the clock to be defined as an open drain and used with external pull up resisters.

The main difference between handshake and normal clock mode is the clock edge that the data is clocked in/out on (See Figure 5-2). In the normal mode, the clock is low, and on the rising edge data is clocked in. The falling edge of the clock is used to change the data. In the handshake mode, this is inverted. The clock is high, and on the falling edge the data is clocked in. The rising edge is used to change the data.

If the PTS is used to transmit data, there needs to be some signal that tells the transmitter that the receiver has received the transmitted data. If there were no signal, the PTS would continuously send data out, regardless if the receiver acknowledges the information or not.

For this reason, the handshake mode was implemented. When the transmission of data is complete, the master clock is left to float (this is why the clock MUST BE setup as open drain output). The receiver, upon reception of the last bit, will pull its clock pin low and hold it low until the SSIO__STBx register is read by the receive processors program. This acknowledges reception of data and that the SSIO__STBx register is ready to receive information.

The master SSIO channel will sense the release of the clock line by the receiver, and start the next transmission.

```
;*****************************************************************
;
; KR SSIO Master Transmit/Slave Receive Handshake
;*****************************************************************
;
P6REG    EQU     0D5H:BYTE          ; window to 1fd5H
P6IO     EQU     0D3H:BYTE          ; window to 1fd3H
P6SSEL   EQU     0D1H:BYTE          ; window to 1fd1H
BAUD     EQU     0B4H:BYTE          ; window to 1fb4H
SSCR_0   EQU     0B1H:BYTE          ; window to 1fb1H
SSCR_1   EQU     0B3H:BYTE          ; window to 1fb3H
Ssio_Stb0 EQU    0B0H:BYTE          ; window to 1fb0H
Ssio_Stb1 EQU    0B2H:BYTE          ; window to 1fb2H
RESULT   EQU     122H:BYTE          ;
WSR      EQU     014H:BYTE          ;
; Send Data from master to slave in Handshake mode
    CSEG            AT      2080H   ;
    LDB WSR, #1FH           ; set window
    LDB P6REG, #0F0H        ; turn off pulldn
    LDB P6IO, #0F0H         ; set as inputs
    LDB P6SSEL, #0F0H       ; special function
    LDB BAUD, #080H         ; 2 Meg Baud
    LDB SSCR_0, #0D9H       ; master_transmit control
    LDB SSCR_1, #18H        ; slave_receive control
    LDB SSIO_STB0, #55H     ; data placed in xmit register
```

270873–29

```
D_WAIT:
    JBC SSCR_1,0,D_WAIT  ; wait for data received
    STB SSIO_STB1, RESULT
    Sjmp        $
```

270873–30

**Program 6. SSIO, Send Byte in Handshake Mode**

The following example uses open drain on both data and clock lines (only clock is required to be open drain in order to function in the handshake mode). The two SSIO channels are still in loop back mode.

Note that in Example 1 the push-pull and open drain modes were used. After reviewing the code from both Examples 1 and 2 the only difference is the set up of the P6IO register and SSIO Control Registers 0 and 1.

## 5.4 SSIO and the PTS

In Example 3 the Peripheral Transaction Server (PTS) and the SSIO moves data from the SSIO receive register to a block of memory using the PTS. Since the PTS is used to transmit and receive data, the handshake mode is used. (SSIO0 and SSIO1 are in loop back mode).

*For reasons previously described, it is a good programming practice to use the handshake mode when using the PTS with the SSIO peripheral.*

```
$include (Macro.kr)
    SSIO0           EQU     1FB0H:BYTE
    SSIO1           EQU     1FB2H:BYTE

    RSEG            AT      30H     ; PTS control Block
PTS_SSIO_0_CNT:     DSB     1       ; Count value
PTS_SSIO_0_CTL:     DSB     1       ; Control for PTS
PTS_SSIO_0_SRC:     DSW     1       ; Source Pointer
PTS_SSIO_0_DST:     DSW     1       ; Dest pointer
SOURCE_BYTE:        DSW     1       ; source data

PTS_SSIO_1_CNT:     DSB     1       ; PTS control Block
PTS_SSIO_1_CTL:     DSB     1       ; control value
PTS_SSIO_1_SRC:     DSW     1       ; source pointer
PTS_SSIO_1_DST:     DSW     1       ; dest pointer
DEST_BYTE:          DSW     1       ; destination byte

    CSEG            AT      2052H
    dcw PTS_SSIO_0_CNT      ; points to PTSCB for SSIO0
    dcw PTS_SSIO_1_CNT      ; points to PTSCB for SSIO1

    CSEG            AT      2032H
    dcw SSIO_0_ISR          ; software ISR for SSIO0
    dcw SSIO_1_ISR          ; software ISR for SSIO1
;
;  Main Program
;     Send data from the source byte to the destination
;     byte trough the SSIO channels using the PTS.
;
;     SSIO 0 sends the information, SSIO 1 receives it.
;     Handshake mode is used in order not to loose any
;     transmitted information.
;
    CSEG            AT      2080H
SSIO_START:
    Ld  SP, #500H           ; SP = Code RAM
    LdB WSR, #1FH           ; Open window 1FH
```

270873–31

```
      Ldb  SSIO_BAUD, #10100111B       ;SSIO BAUD=50k
      Ldb  P6REG, #0F0H                ;turn on pullup
      Ldb  P6IO, #11010000b            ;OD = all but SD0
      Ldb  P6SSEL, #0F0H               ;special FUNC
      ldb  SSIO_STCR0, #11011101B      ;mst_reXmit_Hand
      ldb  SSIO_STCR1, #00011100B      ;slv_rexmt_Hand

      LDB PTS_SSIO_0_CNT, #0FFH        ;send 255 bytes
      LDB PTS_SSIO_0_CTL, #90H         ;single xfer mode
      LD  PTS_SSIO_0_SRC, #SOURCE_BYTE
      LD  PTS_SSIO_0_DST, #SSIO0       ;dest = SSIO0

      LDB PTS_SSIO_1_CNT, #0FFH        ;send 255 bytes
      LDB PTS_SSIO_1_CTL, #90H         ;single xfer mode
      LD  PTS_SSIO_1_SRC, #SSIO1       ;src = SSIO1
      LD  PTS_SSIO_1_DST,#DEST_BYTE

      ClrB WSR        ; clear the window

 ; enable SSIO_0 and SSIO_1 interrupts and PTS
      LdB  INT_MASK1, #00000110B
      Ld   PTS_SELECT, #0600H

 ; send first byte to start the process
      StB  Source_byte, SSIO0

      EI              ; enable interrupts
      EPTS            ; enable PTS
Forever:
      Br   Forever    ; let interrupts take over
;
; Software Interrupt service routines
;
SSIO_0_ISR:
      PUSHA
;
; Because of a BUG with A-step silicon, the PTS
; can interrupt a normal interrupt latency and cause
; an illegal vector to this interrupt service routine
;
; The PTS_SELECT bit must be checked to see
; if this routine has been entered correctly.
;
      JBS  PTS_SELECT+1, 1, Abort_ssio0

Legal_ISR_Vector:
      OR   PTS_SELECT, #0200H          ; turn on the PTS
      LdB  PTS_SSIO_0_CNT, #0FFh       ; and set count

      POPA
      RET

Abort_ssio0:
      ORB INT_PEND1, #02H ; make PTS happen (BUG)
      POPA
; and return
      RET
;
; ISR for other channel
;
SSIO_1_ISR:
      PUSHA
      JBS  PTS_Select+1, 2, Abort_ssio1 ; check bug

Legal_vector:
      OR   PTS_SELECT, #0400H          ; turn on the PTS
      LdB  PTS_SSIO_1_CNT, #0FFh       ; and set count

      POPA
      RET
```

270873–32

```
Abort_ssio1:
      ORB INT_PEND1, #04H ; make PTS happen (BUG)
      POPA
; and return
      RET
end
```

270873–33

**Program 7. SSIO and the PTS**

In the above example several things should be noted. Data is sent out the SSIO0 and received in SSIO1 at 50K baud.

Channel 0 is setup to transmit the first byte, then wait for the receive channel to read the SSIO__STB1 to transmit the next byte (this is accomplished through the SSIO Handshake mode and PTS interrupts on both XFR0 and XFR1).

The PTS is sending information from the SOURCE__BYTE to the DEST__BYTE at 50k baud.

After 255 transfers and receptions, a normal software interrupt is generated. This software interrupts just starts the process all over again.

The most recent data sent and received will be in SOURCE__BYTE and DEST__BYTE.

One way to delay the processor from sending out another byte would be to NOT use the PTS to receive the byte. The transmit could still be used with the PTS, but the next byte would not be sent until the first byte is received. (Also using the handshake mode).

Using the normal software interrupts to receive the information and the PTS to send the information means that the processor can still be left to do other processing with higher baud rates of communication.

## 5.5 Top 5 Issues with the SSIO

(1) Sending Data

1. Setup Port 6
2. Set SSIO__BAUD
3. Set Control with STE bit set
4. Send by writing to STB

(2) Receiving Data
1. Setup Port 6
2. Set SSIO__BAUD
3. Set Control with STE bit set
4. Wait for reception.

(3) Using the PTS with high baud rates will require a great deal CPU time.

(4) Initialize both the TBS and OUF bits accordingly when writing to the SSIO__STCRx registers.

(5) When using the PTS, use the Handshake Mode only.

## 6.0 ANALOG TO DIGITAL CONVERTER

The analog to digital converter on the 8XC196KR is very versatile. It can perform an 8-bit or 10-bit conversion, perform voltage threshold detection, and has two test modes for converting on Analog GND and $V_{REF}$.

There are 8 input channels which are multiplexed to the converter. The sample and conversion times for any channel are programmable, allowing the part to perform fast convertions at any frequency of operation.

The A/D has 4 dedicated SFRs which control its operation. These are the AD__COMMAND, AD__TIME, AD__TEST, and AD__RESULT registers, located at 1FACh, 1FAFh, 1FAEh, and 1FAAh respectively.

## 6.1 A/D Command Register (AD__COMMAND)

Figure 6-1 shows a diagram of the AD__COMMAND register. The lower three bits select the channel to be converted. Bit 3 determines when the conversion should start. If cleared, the conversion will be started by the Event Processor Array (EPA), if set the conversion will start within 3 state times of writing to the AD__COMMAND register.

Bits 4 and 5 are the mode bits. Bit 4's function is dependant upon whether the A/D is performing a normal conversion, or is in threshold mode. If the A/D is performing a regular conversion, bit 5 should be cleared and bit 4 will determine 8- ("1") or 16-bit ("0") conversions.

With bit 5 set, the threshold mode is selected. Clearing bit 4 tells the A/D to detect when the value is above the threshold value; setting bit 4 indicates a value below the threshold should be detected.

The upper 2 bits of the AD__COMMAND are reserved and should be written as "00" to maintain compatibility with future products.

If the AD__COMMAND register is written while a conversion is being performed, the old conversion will be aborted and a new one will start. The A/D done bit in the AD__RESULT register will NOT be set until the completion of the second conversion.

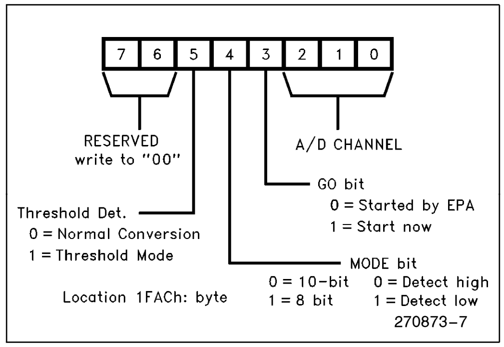



**Figure 6-1. AD__COMMAND Register**

## 6.2 A/D Time Register (AD__TIME)

The AD__TIME register, shown in Figure 6-2, allows the sample window and conversion time (per bit) to be optimized by the user depending on the clock speed of the '196KR. The Sample time controls how long the analog input voltage is connected to the sample capacitor. It must be long enough to properly charge the sample capacitor, but if it is too long, the input voltage may change and introduce error. The Conversion time defines the length of time to convert one bit. It needs to be long enough to allow the comparitor to settle, but cannot be to long or the sample capacitor will discharge and introduce error.

The upper 3 bits of this register program the sample time (SAM), while the lower 5 bits program the conversion time (CONV).

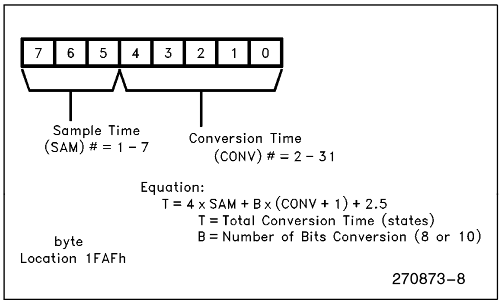



**Figure 6-2. AD_TIME Register**

Sample time (in states) =
4*SAM + 1
Conversion time (in states) =
# of bits * (CONV+1) + 1.5

In order to guarantee an A/D specification of ±3LSBs, the sample time should be at least 3.5μS and the conversion time should be at least 16.5μS. (Consult the latest specs for the most current values.)

The AD_TIME register could be programmed to do a conversion in less than 4us, but the results would be about 200mV off.

Valid ranges for the SAMple window are 1-7 and the CONVersion timer should be between 2 and 31 inclusive. This yields a valid range for the AD_TIME register of 22h-FFH. The AD_TIME register should never be written with all zeros.

Assuming that the sample time is set such that the sample capacitor charges properly. The accuracy will still be a function of conversion time as show in Figure 6-3. This graph was obtained by testing several (typical) devices across automotive temperature range (−40° to +125°). Note that the conversion accuracy drops off VERY rapidly for conversion times under 10μS.

As an example consider an 8XC196KR running at 16Mhz. This gives a state time of 125ns, which means that to meet minimum specs on a 10-bit conversion,the SAM should be programmed with a 7h and CONV should be 0Dh (EDh should be placed in AD_TIME.)

*Note: When determining the sample time, it is extremely important for the user to consider the input circuitry associated with the channel being converted. The circuitry must be able to supply enough current to charge a 2 pF capacitance with a 1μA leakage current in the specified time.*

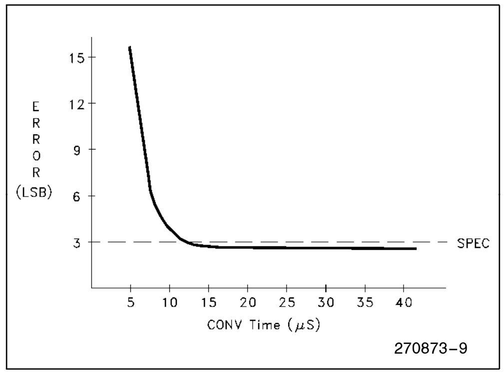



**Figure 6-3. A/D Error vs. Conversion Time**

## 6.3 A/D Test Register (AD_TEST)

The AD_TEST register, shown in Figure 6-4, can be used to enable two test modes, and to modify the zero offset of the A/D. Conversions can be performed on either Analog Ground or $V_{REF}$, to gain insight as to the transfer function of the A/D. Small amounts of zero offset error can be corrected by using the offset adjustment.

*For compatibility with future products, the command register should be written to select channel 7 when performing conversions on $V_{REF}$ or $A_{GND}$.*

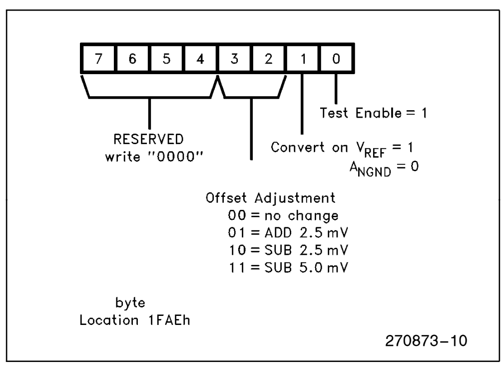



**Figure 6-4. AD_TEST Register**

Bit 0 of AD__TEST is the test enable bit. It should be clear to perform normal conversions on any of the eight analog inputs, and set to perform conversions on $V_{REF}$ or $A_{GND}$.

Bit 1 selects between $V_{REF}$ ("1") or $A_{GND}$ ("0"). Bits 2 and 3 control the offset adjustment as shown in Figure 6-4. Later a sample program will be presented which demonstrates how to adjust the offset. Small amounts of offset errors, both negative and positive, can be adjusted through these bits. When the AD__RESULT register is read, the effected result (after offset adjustment) will be reflected, automatically.

The upper four bits are reserved and should be written as all 0's.

Figure 6-6 shows a typical low temperature (−40° C) graph of absolute error vs input voltage. This depicts many different types of errors inherent in the conversion process. The zero offset error can be seen, about 4mV, as well as the full-scale error, about 8mV.

Another effect which generally only shows up at low temperatues is a "bowing" S curve that can be seen across the entire transfer function. This is caused by small amounts of noise accumulated across the entire resister ladder.

At higher temperatures, bowing dissappears, leaving only the stair-step or saw tooth effect. This is usually caused by errors in the resister ladder when the values "foldback" to conserve space.

## 6.4 A/D Result Register (AD__RESULT)

The last SFR is the AD__RESULT register shown in Figure 6.5. The lower three bits in this word register contain the channel number that was converted. Bit 3 is the BUSY bit and is set approximately 8 state times after a conversion is started. Bits 4 and 5 are reserved (ignore read value).

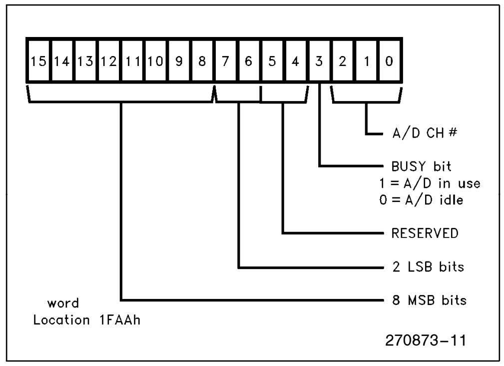



**Figure 6-5. AD__RESULT Register**

The remaining 10 MSB bits contain the result of the conversion. If an 8-bit conversion was performed, the result will be stored in the MSBs (bits 8-15) of AD__RESULT. The lower two bits (bits 6 and 7) are undefined.

In threshold detection mode, the upper 10-bits of AD__RESULT should be programmed with an 10-bit value which serves as the threshold. Once threshold mode is entered, continuous conversions are performed on the selected channel until the desired threshold crossing occurs (5mV resolution). Conversions will then stop and an interrupt pending will be issued.

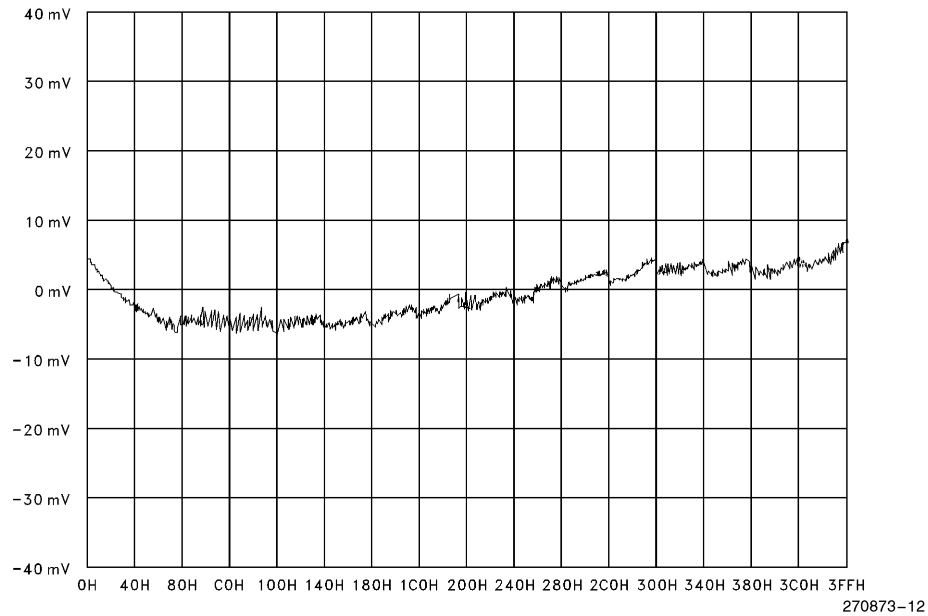



**Figure 6-6. A Typical A/D Transfer Function Error, with Offset and Full Scale Errors**

```
R0              EQU  00h:WORD   ;00 register
AD_Time_W       EQU  0EFh:BYTE  ;Window to AD_Time register
AD_Test_W       EQU  0EEh:BYTE  ;Window to AD_Test register
AD_Result_W     EQU  0EAh:WORD  ;Window to AD_Result register
AD_Command_W    EQU  0ECh:WORD  ;Window to AD_Command register

        CSEG AT     2080H

TI_START:

        LdB   WSR,#7Dh                ;window 1FA0-1FBF to E0-FF
        LdB   AD_Time_W,#0EFh         ; 3.5 us samp/20us convert
        LdB   AD_Test_W,#00h          ;normal conversion

        LdB   AD_Command_W,#0Ch       ;start conversion, ADCH5
        Ld    R0,R0                   ;dummy statements to wait for
        Ld    R0,R0                   ;BUSY bit to get set
wait:
        JbS   AD_Result_W,3,wait      ;wait for conversion
        St    AD_Result_W,80h         ;save it
```

270873–34

**Figure 6-7. Program Segment to Initialize A/D and Convert on ACH5**

## 6.5 Example A/D Programs

In this section, a few examples will be presented on using the A/D. The first program segment shows how to start a 10 bit conversion on channel 5. The AD__TIME register is set-up, followed by writing to the AD__COMMAND register. The AD__RESULT register is then polled until the conversion is complete.

### 6.5.1 USING THE A/D WITH THE PTS

The Peripheral Transaction Server (PTS) can be used with the A/D, allowing up to 256 conversions without CPU intervention. In the example program below, the PTS is used to perform 8 10-bit conversions (one for each channel).

The A/D scan mode in the PTS is used to perform this function. It operates as follows:

1. The word pointed to by PTS__SD is read. The lower byte is saved in a temporary location within the ALU.
2. PTS__SD pointer is incremented by two.
3. The register pointed to be PTS__REG is read and stored at the location pointed to by PTS__SD.
4. PTS__REG is then incremented by two.
5. The value that was saved in (1) is stored in the register pointed to by PTS__REG.
6. PTS__SD is optionally updated.
7. PTS__COUNT is decremeneted.

When PTS_COUNT reaches zero, the PTS_SELECT bit for the A/D is cleared and the PTS_SRV bit for the A/D is set causing a normal interrupt to happen.

The PTS_REG is set up to point to the AD_RESULT register (It can only point to Register RAM or SFR locations), thus it is read every PTS cycle, and when PTS_REG is incremented by two, it points to the AD_COMMAND register which is then written.

Since the last thing that the PTS cycle does is a load into the AD_COMMAND register (to start another conversion), the last value in the A/D result RAM table should be loaded with 0000h. This will NOT start an A/D conversion when the last channel is complete (Remember that the process starts with a conversion started manually).

The table format is shown in Figure 6-8:

| | |
|---|---|
| AD_RESULT for ACH0 : WORD | |
| DUMMY COMMAND "00" : WORD | |
| AD_RESULT for ACH1 : WORD | |
| AD_COMMAND for ACH0 : WORD | |
| AD_RESULT for ACH2 : WORD | |
| AD_COMMAND for ACH1 : WORD | 16 WORDs |
| AD_RESULT for ACH3 : WORD | of |
| AD_COMMAND for ACH2 : WORD | A/D results |
| AD_RESULT for ACH4 : WORD | and |
| AD_COMMAND for ACH3 : WORD | A/D commands |
| AD_RESULT for ACH5 : WORD | |
| AD_COMMAND for ACH4 : WORD | |
| AD_RESULT for ACH6 : WORD | |
| AD_COMMAND for ACH5 : WORD | |
| AD_RESULT for ACH7 : WORD | |
| AD_COMMAND for ACH6 : WORD | PTS_SOURCE |

AD_COMMAND for ACH7 is done manually to start the scan

**Figure 6-8. Example A/D Scan Mode Table**

The routine starts off by initializing AD_TEST and AD_TIME. The PTS control block is then setup with PTS_REG pointing to AD_RESULT and PTS_SD pointing to the start of the table. The next block of code sets up the table by filling all of the A/D command slots.

For simplicity in coding, each A/D channel is done in succession (7 $\geq$ 0). However, any order conversions can be performed, as well as any bit combination (10- or 8-bit conversions).

Interrupts for the A/D are masked (enabled) in both the core and PTS. Then the A/D is started by writing to the AD_COMMAND register (starting an A/D on channel 7).

After the first sample is completed, the PTS reads the result and stores it in a table. It then loads the next command, from the table, into the AD_COMMAND register. This is repeated until 8 values have been read. Note the that last command (for the 8th read/write) is a "dummy" command and does not start another A/D conversion, as the A/D doesn't need to be restarted.

After 8 samples are collected, the A/D interrupt service routine is called and the data table is "cleaned up" by shifting all of the data words to the right by 6. This leaves just the sample value in the data table.

```
$nolist
$include(kr.inc)
$list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; This code samples A/D channels 0-7 is sucession using
;   the PTS scan mode.  The data is stored in table, as follows:
;
;  A/D_Command1    A/D_Data0    A/D_Commad2    A/D_Data1 ....
;  A/D_Command7    A/D_Data6    0000           A/D_Data7
;
; The "0000" in place of the last command for the A/D is needed
; so that another, unwanted, conversion will not be started.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
AD_Time_W         EQU   0EFh:BYTE        ;Window to AD_Time register
AD_Test_W         EQU   0EEh:BYTE        ;Window to AD_Test register
AD_Result_W       EQU   0EAh:WORD        ;Window to AD_Result register
AD_Command_W      EQU   0ECh:WORD        ;Window to AD_Command register

TBL               EQU   082h:WORD        ;Pointer into command/data table
TEMP              EQU   084h:WORD        ;Temporary storage
TABLE             EQU   3000h:WORD ;start of command/data table

        CSEG  AT  204Ah               ;Pointer to AD PTS command block
        DCW       PTSCOUNT

        CSEG  AT  200Ah               ;Set up AD_ISR pointer
        DCW       AD_INT

        RSEG  AT  070h                ;allocate AD PTS command block
PTSCOUNT:         DSB   1
PTS_CONTROL:      DSB   1
PTS_SD:           DSW   1
PTS_REG:          DSW   1

        CSEG  AT  2080H
TI_START:   DI
            DPTS
            LdB   WSR,#7Dh               ;window 1FA0-1FBF to E0-FF
            LdB   AD_Time_W,#0EFh        ;3.5us samp/20us convert
            LdB   AD_Test_W,#00h         ;normal conversion
;
;Set up PTS block for AD
;
            LdB   PTSCOUNT,#08h          ;8 total conversions
            LdB   PTS_CONTROL,#0CBh      ;A/D scan; SU,SI,DI
            Ld    PTS_SD,#Table          ;point to data table
            Ld    PTS_REG,#AD_Result     ;point to AD_result SFR
```

270873-35

**Program 8a. A/D Scan Mode using the PTS**

```
;
;Fill Command table
;
        Ld    TBL,#TABLE
        Ld    TEMP,#000Eh       ;Command for starting conversion, channel 6
setup:  St    TEMP,[TBL]+       ;fill command bytes in table
        Inc   TBL               ;skip data word in table
        Inc   TBL               ; (result word)
        Dec   TEMP              ;decerement command by 1
        Cmp   TEMP,#0007h
        Jne   setup             ;stop when TEMP = 0007h

;NOTE: After the last channel is read by the PTS, the A/D should NOT
; be restarted. Therefore the last command entry in the table should
; be a dummy and not start a conversion!
        St    R0,[TBL]                ;No new conversion
        OrB   INT_MASK,#20h           ;enable AD done interrupt
        OrB   PTS_SELECT,#20h         ;enable AD done to PTS
        LdB   AD_Command_W,#0Fh             ;start conversion, ADCH7
        EI
        EPTS
self:   Sjmp  self
;
; A/D Interrupt Service Routine
AD_INT:
        PUSHA
        JBS   PTS_SELECT,5,abort              ;check validity of enterance
;The following code "cleans up" the data from the AD.
;The lower six bits are discarded, leaving only the 10 data bits.
;
        Ld    TBL,#TABLE+2            ;start of data
        LdB   80h,#08h                ;count
fix:    Ld    TEMP,[TBL]
        Shr   TEMP,#06h                       ;get rid of junk
        St    TEMP,[TBL]+
        Inc   TBL                                 ;skip command word
        Inc   TBL
        Djnz  80h,fix
        POPA
        RET
;If we got here due to the bug in the interrupt handler
;force a call to the PTS routine, as this should have been called
;in the first place.
abort:  OrB   INT_PEND,#20h           ;force call to PTS
        POPA
        RET
    end
```

270873–36

**Program 8b. A/D Scan Mode using the PTS**

## 6.6 Threshold Detection

The threshold mode on the A/D allows the CPU to be notified when the value on one of the A/D channels crosses either above or below a predetermined value. The following code demonstrates one possible application of this A/D mode.

The A/D is set up to monitor a channel and generate an interrupt when the value on it passes above 2.5 V. The conversion is started and the 8XC196KR is put into idle mode. When the A/D determines that the value on ADCH0 is greater than 2.5 V, an interrupt is generated and the CPU exits the idle mode.

```
;The window equates are the same as before

     CSEG     AT  200AH
     DCW  AD_ISR

     CSEG     AT  2080H
TI_START:
     DI
     OrB INT_MASK,#20h    ;enable a/d interrupt
     LdB WSR,#7Dh         ;1FA0-1FBF to E0-FF
     LdB AD_Time_W,#0EFh; 3.5us samp/20us convert
     LdB AD_Test_W,#00h   ;normal conversion
     EI
; Write to RESULT (Ach0 > 2.5 volts)
; Upper 10-bits the RESULT sets the threshold value
     Ld  AD_Result_W,#7FC0h
     Ldb AD_Command_W, #28h
     Idlpd  1      ;enter idle, wake up when a/d done
;
; continue executing when A/D sees >2.5v on ADCH0
```

270873-37

When the A/D conversion value crosses above the 2.5 volt level, set by the upper byte of the AD__COMMAND register. The device will exit idle mode and execute the AD__ISR interrupt service routine.

## 6.7 A/D Test Modes

The A/D on the 8XC196KR can perform conversions on $A_{GND}$ or $V_{REF}$. This allows for the software detection of offset and full-scale/gain errors. Small amounts of offset errors can be adjusted by writing to the AD__TEST register (bits 2/3).

The following code segment attempts to minimize the offset error. The method used is very straight-forward. The routine is called by the main routine. It starts by assuming that a −5.0mv offset is the best adjustment for offset correction. It then performs 16 conversions on $A_{GND}$, summing the results. If the sum is more than 8 then the next higher offset value is tried. This is repeated until an offset with less than 8 on 16 conversions is found, or, if none can be found, +2.5mv is used. The code will return to the main routine with the best offset value in the AD__TEST register.

```
;The window equates are the same as before
sum       EQU 070h:WORD  ; sum of conversion results
count     EQU 072h:BYTE  ; count for conversions
temp      EQU 074h:WORD  ; result of last conversion

     CSEG
AD_OFFSET_ADJUST:
     LdB       INT_MASK,#00h  ; disable A/D interrupts
     LdB       WSR,#7Dh       ; 1FA0-1FBF to E0-FF
     LdB       AD_time_w,#0efh; 23.5us convert
     LdB       AD_Test_W,#0Dh; Convert Agnd -5.0mv
     Scall     check
     LdB       AD_Test_W,#09h ; Convert (Agnd-2.5mv)
     Scall     check
     LdB       AD_Test_W,#01h ; Convert (Agnd+0.0mv)
     Scall     check
     LdB       AD_Test_W,#05h ; +2.5mv offset wins
     Sjmp      exit
check:
     Clr       sum            ; Clr SUM
     LdB       count,#10h     ; perform 16 conversions
conv:
     LdB       AD_command_w,#0fh ;10-bit con. on ch7
     Ld        R0,R0          ;dummy to wait for
     Ld        R0,R0          ;BUSY bit to be set
wait:
     JBS       AD_result_w,3,wait   ; wait till done
     Ld        temp,AD_Result_W
     Shr       temp,#06h      ; just want result
     Add       sum,temp       ; sum results
     Djnz      count,conv     ; loop if not done
     Cmp       sum,#08h       ; arbitrary cutoff
     Jh        exit
; if sum is less than 8, then return to main routine by
; POPing the old Program Counter off the stack and
; returning
     Pop       temp           ; return to main
exit:
     RET
```

270873-38

TIMER1 = 1F98H:Byte

TIMER2 = 1F9CH:Byte

**TIMERn__CONTROL**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| CE | UD | M2 | M1 | M0 | P2 | P1 | P0 |

**CE** - "0" Disable Timer, "1" Enable Timer

**UD** - "0" Count Down, "1" Count Up

**M2,M1,M0**

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | Clock = Xtal1, Direction = UD |
| x | 0 | 1 | Clock = TxClk, Direction = UD |
| 0 | 1 | 0 | Clock = Xtal1, Direction = TxDir |
| 0 | 1 | 1 | Clock = TxClk, Direction = TxDir |
| 1 | 0 | 0 | Clock = T1 Overflow, Direction = UD |
| 1 | 1 | 0 | Clock = T1 Overflow, Direction = T1 |
| 1 | 1 | 1 | Quadrature Counting, TxClk & TxDir |

**P2,P1,P0**

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | %1 (or Xtal/4 - 250nS at 16MHz) |
| 0 | 0 | 1 | %2 (or Xtal/8 - 500nS at 16MHz) |
| 0 | 1 | 0 | %4 (or Xtal/16 - 1μS at 16MHz) |
| 0 | 1 | 1 | %8 (or Xtal/32 - 2μS at 16MHz) |
| 1 | 0 | 0 | %16 (or Xtal/64 - 4μS at 16MHz) |
| 1 | 0 | 1 | %32 (or Xtal/128 - 8μS at 16MHz) |
| 1 | 1 | 0 | %64 (or Xtal/256 - 16μS at 16MHz) |
| 1 | 1 | 1 | RESERVED |

**Figure 7-1. TIMER__CONTROL Register**

## 6.8 Top 5 Issues with the A/D

(1) Small amounts of offset errors can be adjusted using the AD__TEST register.

(2) AD__COMMAND and AD__RESULT are two (2) separate SFRs.

(3) Threshold Detect Mode of the A/D does a continuous conversion on one channel.

(4) Conversion times of less than 16uS and Sample times of less than 3.5uS will not produce less than ± 3 LSB error.

(5) In the AD__TEST register, the LSB bit is the ENABLE bit for the Convert on $A_{GND}$ or $V_{REF}$. Bit 1 is the control bit for which to convert (1 for $V_{REF}$ 0 for $A_{GND}$).

## 7.0 EVENT PROCESSOR ARRAY (EPA)

### 7.1 Timers

The EPA has two 16-bit timers/counters, which are controlled by two registers, TIMER1__CONTROL and TIMER2__CONTROL at 1F98h and 1F9Ch, respectively. The values of the timers can be read at 1F9Ah and 1F9Eh (TIMER1 and TIMER2). The maximum count rate of each timer is based on the internal clock rate divided by 4. This yields a resolution of 250ns at 16MHz. In addition, TIMER2 can be programmed to count when TIMER1 over/under flows, allowing a 32- bit counter to be formed.

*Although the EPA can not capture or compare on 32-bit values, an EPA channel can be programmed to interrupt on any number of TIMER1 overflows. In addition, TIMER2, with the increment on overflow, can still be prescaled.*

Both timers can be clocked by an internal or external clock with internal or external direction (Up/Down) control. Quadrature clocking is also available, allowing for easy interfacing with an encoder wheel.

*The timers do not have auto-reload features, but encoder wheel interface can be acheived through a small amount of PTS interface. This will not be demonstrated in this paper.*

Timer overflows are mapped into the EPAINTx interrupt in the core. However, each timer has its own mask and interrupt pending bit in the EPA__MASK1 and EPA__PEND1 registers.

Figure 7-1 shows a map of the TIMERn__CONTROL register. Bit 7 is the timer enable bit, and controls whether the timer is active. Bit 6 is the internal direction control (Up = 1, Down = 0).

The next three bits (bits 5–3) are the mode control bits. They determine if the clock and direction control should be internal or external, or if quadrature clocking should be used.

The three least significant bits control the prescale to be applied to the clock. These range from a divide by 1 (internal clock/4) to a divide by 64 (internal clock/256). This prescale is applied to both internal AND external clocks. *NOTE: When using an external clock, the timer will count on EACH edge of the clock (assuming no prescaling is in effect).*

The time registers (TIMER1 and TIMER2) are both readable and writable. This allows for more flexibility in the generation of interrupts.

### 7.1.1 TIMER EXAMPLES

The code segment below shows how to set up a software timer which will generate an interrupt in 1 mS. TIMER1 is first loaded with 1000 (decimal). Then it is programmed to count down, once per microsecond. The EPA_MASK1 is set up to allow interrupts on TIMER1 overflows or underflows.

The interrupt occurs when the TIMER1 rolls under 0000. The interrupt service routine for EPAINTx will have to determine which source caused the interrupt and take whatever action is needed. An example of this is given later in the EPA outputs section.

```
temp    equ     070h:word

Cseg at 2080h

Ld      temp,EPA_MASK1
Or      temp,#02h
St      temp,EPA_MASK1  ;enable Timer1 overflow int
OrB     INT_MASK,#01h   ;enable EPAINTx in core
Ld      temp,#1000
St      temp,TIMER1             ;count 1000 times
LdB     temp,#082h              ;count down
StB     temp,TIMER1_CONTROL     ;at 1 count per us

Br      $                       ;wait for underflow
```

270873–39

## 7.2 EPA Input/Output Structure

The EPA section has ten (10) Capture/Compare modules which each support timed event input and output for a single pin. There are also two Compare only modules (COMP0 and COMP1) which share their outputs with two of the EPA channels (EPA8 and EP9 respectively). The Capture mode can be used to generate an interrupt on an input edge, reset the opposite time base timer, start an A/D conversion, or simply capture the time a transition of its input pin occurred.

The Compare function is for output time events. It can change the state of its output pin when its time base timer matches the value in its EPA_TIMEn register. Also, an EPA channel has the option of resetting its own timer as well as the opposite timer, or start a timed A/D conversions.

There are two dedicate SFRs for each EPA channel that control the operation. These are EPA_CONTROLn, and EPA_TIMEn registers. The EPA_CONTROLn is detailed in Figure 7-2.

The EPA_CONTROLn set of registers are used to configure their associated pin. The bit map of the control register is as follows:

**EPA_CONTROLn**

| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| RM | TB | CE | M1 | M0 | RE | AD | ROT | ON/RT |

RM - "1" Enables Remapping (EPA1 and EPA3 Only)

TB - "0" = Timer1, "1" = Timer2

CE - "1" Enables Comparator

| M1,M0 | Capture | Compare |
|---|---|---|
| 0 0 | No Op | Interrupt Only |
| 0 1 | Capture − edge | Output "0" |
| 1 0 | Capture + edge | Output "1" |
| 1 1 | Capture +/− edge | Toggle Output |

RE - "1" = Lock Time Entry

AD - "1" = Start A/D Conversion

ROT - "0" = Same Timer as TB, "1" = Opposite

ON/RT - Overrun/Reset Timer Enable

**Figure 7-2. EPA_CONTROL Register**

**RM** Bit 8 of EPA_CONTROL. This only has an effect on channels 1 and 3. Setting this bit enables the lower adjacent channel to set/reset/toggle the channels output pin. This allows channels 0 and 1 to control the same output (channel 1's), or channels 2 and 3 to control output on channel 3. Clearing this bit disables the remap feature.

*EPA_CONTROL1 and EPA_CONTROL3 must be written as WORDs.*

**TB** Bit 7 is used to select which timer should be used as the time base for captures or compares. A "0" selects TIMER1, and a "1" selects TIMER2.

**CE** Bit 6 selects between capture and compare modes: "0" selects capture mode while "1" enables the comparator. By enabling the comparator any mode will cause an interrupt. While enabling the capture function, only generates an interrupt when modes 1, 2, or 3 is used.

**Mx** Mode bits. Bits 5 and 4 are the mode select bits. They operate as follows:

**In Capture Mode**

| M1 | M0 | Action |
|---|---|---|
| 0 | 0 | no operation |
| 0 | 1 | capture on + edges |
| 1 | 0 | capture on − edges |
| 1 | 1 | capture +/− edges |

**In Compare Mode**

| M1 | M0 | Action |
|---|---|---|
| 0 | 0 | no operation |
| 0 | 1 | reset output pin |
| 1 | 0 | set output pin |
| 1 | 1 | toggle output pin |

**RE** Bit 3 can be used to "lock" a Compare function. When set to "1", the compare function is always enabled. When clear, the event will occur only once, and then the EPA_TIMEn value must be rewritten.

**AD** Bit 2, when set will start an A/D conversion when a capture or compare event occurs. It has no effect when clear.

**ROT** Bit 1. In Capture, a "1" resets opposite base timer (not TB), while a "0" has no action. In Compare, a "1" selects opposite base timer (not TB) for reset, while a "0" selects the TB bit timer for reset.

**ON/RT** Bit 0. On Captures, a "1" means that the old data can be overwritten on an overrun, while a "0" means that the new data is lost (ignored) on overruns. In Compare, a "1" means that the timer selected by TB and ROT will be reset. A "0" will have no action.

The EPA_TIMEn register has two functions, depending on the mode of the channel. In Capture mode, the register is double buffered and holds the time of the transition (i.e., the value of the selected timer at the instant the transition was detected is saved in the EPA_TIMEn register). If the old time has not been read and the buffer is full when a new transition occurs, an overrun interrupt request will occur.

An interrupt is generated on the load of the EPA_TIMEn value, either from the buffer or directly if the buffer is empty. The EPA_TIMEn value must read each interrupt service in order to obtain more than one interrupt.

In Compare mode, EPA_TIMEn is programmed with the time that events are to occur. Mutiple events can occur per time match. For example, the output pin can be made to set/reset/toggle, and generate internal functions such as starting the A/D and resetting a timer.

*Note: The EPA and Compare CONTROLn registers should be written as words.*

**EPA__PEND (1FA2H:Word), EPA__MASK (1FA0H:Word)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EPA INT4 | EPA INT5 | EPA INT6 | EPA INT7 | EPA INT8 | EPA INT9 | OVR INT0 | OVR INT1 | OVR INT2 | OVR INT3 | OVR INT4 | O VR INT5 | OVR INT6 | OVR INT7 | OVR INT8 | OVR INT9 |

EPAINT4 - Highest Priority
EPAINT5 - .
. .
. .
. .
. .
. .
OVRTMR1 -
OVRTMR2 - Lowest Priority

EPA__PEND1 (1FA6H: Byte), EPA__MASK1 (1FA4H: Byte)

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| rsv | rsv | rsv | rsv | COMP CH0 | COMP CH1 | OVR TMR1 | OVR TMR2 |

**Figure 7-3. The EPA__PEND and EPA__MASK Registers**

## 7.3 EPA Interrupts

There are 24 sources of interrupts within the EPA; 12 event interrupts (10 Capture/Compare and 2 compare only), 10 input overflow interrupts (for overrun errors) and two timer overflow interrupts.

Interrupts can be masked either in the core (for events on channels 0-3), or in the EPA mask registers for all other sources. See Figure 7-3 for bit map of EPA__MASK, and EPA__MASK1. All channels excluding EPA0-3 share a common interrupt request line to the core, via EPAINTx. This means that EPA channels 0-3 can have their interrupt requests serviced directly, while all other sources must be decoded.

The decoding is handled by the use of the EPAIPV (EPA Interrupt Priority Vector) which is located at 1FA8h. This register always contains a code for the next highest priority interrupt which is both pending and masked. All EPAIPV values are shown in FIGURE 7-4.

This register is designed to be used in conjunction with the TIJMP instruction to vector to the appropriate interrupt service routine. The EPAIPV should be read until it returns 00h before exiting the EPAINTx service routine. This insures that all pending and masked interrupts have been acted upon; also this is the only way to clear the EPAINTx pending bit in the INT__PEND and pending bits in the EPA__PEND / EPA__PEND1 registers.

When using the EPAIPV with the TIJMP instruction, some care must be used. The EPAIPV register always returns a number with the LSB clear. (ie 02,04 ...). However, the TIJMP multiplies this by two when calculating the offset into the jump table. The result of this is that consecutive jump vectors will not be consecutive in memory. There will always be an unused word between them.

See the program example 16 in the EPA outputs section for details on using EPAIPV with TIJMP.

The EPA__MASK and EPA__MASK1 registers are only word addressable. Do NOT attempt to write to them as bytes as this will have no effect.

**EPAIPV (1FA8H: Byte)**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | PV5 | PV4 | PV3 | PV2 | PV1 | 0 |

**rsv** Reserved

**PV1-PV5** Encoded Highest Priority Interrupt Values from 2-28H.

| | |
|---|---|
| 0 = No Interrupt pending | 14H = OVRINT4 |
| 28H = EPAINT4 | 12H = OVRINT5 |
| 26H = EPAINT5 | 10H = OVRINT6 |
| 24H = EPAINT6 | 0EH = OVRINT7 |
| 22H = EPAINT7 | 0CH = OVRINT8 |
| 20H = EPAINT8 | 0AH = OVRINT9 |
| 1EH = EPAINT9 | 08H = Compare Channel 0 |
| 1CH = OVRINT0 | 06H = Compare Channel 1 |
| 1AH = OVRINT1 | 04H = TIMER1 Overflow |
| 18H = OVRINT2 | 02H = TIMER2 Overflow |
| 16H = OVRINT3 | |

**Figure 7-4. EPA Interrupt Priority Vector**

```
$include (KR.INC)
        RSEG    at 70h
time:           dsw     1           ;time of conversion
result:         dsw     1           ;result of conversion
temp:           dsw     1           ;temporary storage

        CSEG    at      2008h       ;set up EPA0 ISR pointer
        dcw     EPA0ISR

        CSEG    at      200Ah
        dcw     ADISR               ;set up AD ISR pointer

        CSEG    at      2080h
START:
        LdB     temp,#0EFh
        StB     temp,AD_TIME                ;23.5us conversion
        LdB     temp,#03h
        StB     temp,AD_COMMAND             ;10-bit on Ach3
        LdB     temp,#01h
        StB     temp,P1REG                  ;turn off pull down
        StB     temp,P1IO                   ;P1.0 is an input
        StB     temp,P1SSEL                 ;Select P1.0 for EPA
        LdB     temp,#0C2h
        StB     temp,TIMER1_CONTROL         ;start timer1, 1us per count
        Ld      temp,#24h
        St      temp,EPA_CONTROL0           ;start conv. on + edge
        OrB     INT_MASK,#30h               ;enable a/d, epa0 interrupts
        Ei
        Br      $                           ;let epa/interrupts do the work

EPA0ISR:
        PUSHA
        Ld      time,EPA_TIME0  ;save time of conversion
        POPA
        RET

ADISR:
        PUSHA
        Ld      result,AD_RESULT ;save conversion result
        POPA
        RET
```

270873–40

**Program 9. Start an A/D Conversion on a Positive Input Edge**

## 7.4 Input Capture

The Capture modules of the EPA can be used, among other things, to time-stamp events which occur on EPA input channels. When an event occurs, the value of the selected timer is loaded into the EPA__TIME registers associated with the channel on which it occurred. If the EPA__TIME register is full, then the data is buffered. However, if the buffer is also full, then either the new data, or the data in the buffer will be lost, depending on the state of the ON/RT bit in the EPA__CONTROLn register.

An interrupt is generated each time the EPA__TIMEn register is loaded; whether directly or from the buffer. Therefore, the EPA__TIMEn register must be read even if the data is not being used to allow another interrupt to be generated.

### 7.4.1 HSI EXAMPLE #1

Consider the problem of synchronizing an A/D conversion with a clock pulse. This can be done using an EPA capture channel which is programmed to start an A/D conversion. (See Program 9).

First, the A/D is programmed to perform a conversion on channel 3. The conversion is programmed to be started by the EPA. Next Port baud 1.0 is set up for use by the EPA, and EPA0 is programmed to look for a rising edge.

When the EPA senses a rising edge on channel 0, it stores the value in TIMER1 into EPA__TIME0, and starts the A/D converter.

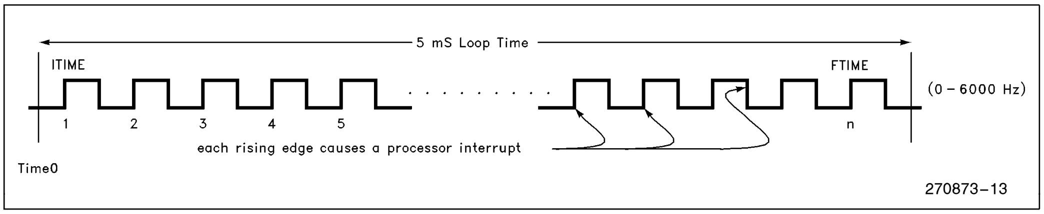



**Figure 7-5. Wheel Speed Signal for each Wheel**

### 7.4.2 HSI EXAMPLE #2 : ABS

One problem in implementing an Anti-Lock Braking System is determining the individual speed of the wheels. The following program 10a shows one way (using both the EPA and PTS) to calculate the speed of a wheel. It is easy to modify the program to perform speed calculations for 4 wheels. Just add in three more EPA channels, and PTS control blocks.

It is assumed that a square wave is fed into an EPA channel whose frequency is proportional to the speed of the wheel. EPA channel 0 is used to look for rising edges. The number of rising edges seen in a five millisecond interval is counted, and this is used to determine the average wheel speed.

The PTS is used to count the number of pulses received, and to determine the first and last time that a rising edge is detected during a five millisecond loop.

Figure 7-5 shows a sample input to the EPA. The following sequence of events takes place to determine the wheel speed:

1. The first edge in the 5ms interval causes a normal EPA interrupt. The time from EPA__TIME0 is saved into ITIME. The PTS is enabled to handle counting the rest of the edges.
2. Edges 2 through "n" cause a PTS cycle to occur. The PTS moves EPA__TIME0 to FTIME and decrements PTSCOUNT. If the PTSCOUNT is initialized to 0FFh, negating PTSCOUNT will yield the number of edges captured by the PTS plus the first edge seen by the normal interrupt for edge 1.
3. The 5ms interrupt disables the PTS channel (so the next rising edge will cause a normal software interrupt repeating the cycle). It also calculates the wheel speed using the following formula:

$$\text{Speed} = \frac{(\text{Ftime-Itime})*\text{conv\_\_fact}}{\text{NEG(PTS\_\_COUNT)}}$$

```
$include(Kr.inc)
; *  SFR'S accessed by window 1F  *
;
P1IO_1F                 EQU     0D2H:BYTE       ; R/W  Window 1FH (1FD2H)
P1SSel_1F               EQU     0D0H:BYTE       ; R/W  Window 1FH (1FD0H)
Comp_Time0_1F           EQU     08AH:WORD       ; R/W  Window 1Fh (1F8AH)
Comp_Control0_1F        EQU     088H:BYTE       ; R/W  Window 1Fh (1F88H)
EPA_Mask1_1F            EQU     0A4H:BYTE       ; R/W  Window 1Fh (1FA4H)
Timer1_Control_1F       EQU     098H:BYTE       ; R/W  Window 1Fh (1F98H)
;
; *  program equates  *
;
Speed_high_Constant     EQU     0007H   ; this is our magic number
Speed_low_Constant      EQU     0A120H          ; in order to get Hz in result using 16MHz
;
; *  general purpose RAM  *
;
        rseg    at      0b0h

; PTS Control Block for EPA0
PTS_Count_EPA0:         DSB     1
PTS_Control_EPA0:       DSB     1
PTS_Source_EPA0:        DSW     1
PTS_Dest_EPA0:          DSW     1
ITime_0:                DSW     1

FTIME_0:                DSW     1       ; Final Time in 5ms window
num_of_pulses_0:        DSW     1       ; number of pulses
inv_speed_0:            DSL     1       ; speed indicator
Templ:                  DSL     1       ; Long temp register
EPAIPV_Ptr:             DSW     1       ; Priority Vector Shadow Reg for TIJMP
EPAIntX_JTBase_Ptr:     DSW     1       ; EPAx Jump Table Base pointer
;
; *  Interrupt Vectors  *
;
            cseg    at      2008H
            DCW     EPA0_ISR
            cseg    at      2048H
            DCW     PTS_COUNT_EPA0
            cseg    at      2000H
            DCW     EPAx_ISR
```

270873–41

**Program 10a. ABS Input Frequency Detection using the PTS and EPA Inputs**

```
;
; *  main routine  *
;
        cseg at 2080h

Initial_Start:
        DI                              ; Disable Interrupts
        DPTS                            ; Disable PTS
        LD      SP,#500h                ; Initialize Stack
        Call    Clear_RAM

Software_Tmr_Init:
        LDB     WSR, #1FH                    ; Use Window 1FH ( 1F80-1FFF -> 80-FF )
        LD      EPA_Mask1_1F,#08h            ; Enable compare module interrupt
        ORB     Int_Mask, #1Fh               ; Enable EPA0-3 & EPAX interrupts
        LDB     Comp_Control0_1F,#40h        ; Enable Software timer0
        LD      Comp_Time0_1F,#2500          ; Initialize periodic 5ms interrupts
        CLRB    WSR                          ; Reset WSR to zero window
        LD      EPAIntX_JTBase_Ptr,#EPAIntX_JTBase ; TIJMP Table Pointer

PTS_Initialize:
        LDB     PTS_Control_EPA0,#80h        ;Single word transfer
        LD      PTS_Source_EPA0,#EPA_Time0
        LD      PTS_Dest_EPA0,#FTime_0

EPA_Initialize:
        LD      Templ,#0FE20H                ; CAPTURE ON POSITIVE EDGES
        ST      Templ,EPA_CONTROL0[0]        ;Initialize EPA_CONTROL0

Port_Configure:
        LDB     WSR, #1Fh                    ; Use Window 1Fh ( 1F80-1FFF -> 80-FF )
        ORB     P1SSel_1F,#01h               ; EPA0 Configure
        ORB     P1IO_1F,  #01h               ; Set as HSI
        LDB     TIMER1_CONTROL_1F,#0C3h           ; Turn on TIMER1
        CLRB    WSR                                              ; Reset WSR to 0
        EI                                   ; Enable Interrupts
        EPTS                                 ; Enable PTS

Loop_Forever:
        SJMP    Loop_Forever                 ; Let interupts take-over

; Foreground processing can use the wheel speed indicators.  The processor
; will be interrupted every five milliseconds for wheel speed calculations.

Clear_RAM:
        Clr     ITIME_0                      ;Clear initial time values
        Clr     FTIME_0                      ;Clear final time values
        RET
;
; *  EPA INTERRUPT SERVICE  *
;
EPA0_ISR:
        PUSHA
        JbS     PTS_Select,4,fix0            ;check for int. bug
        LD      Itime_0,EPA_TIME0            ; store initial time
        LDB     PTS_Count_EPA0,#0FFH         ; set count = 255
        ORB     PTS_Select,#10h              ; Next interrupt will be PTS
        POPA
        RET
FIX0:
        ORB     INT_PEND,#10h                ; force call to PTS
        POPA
        RET
```

270873–42

**Program 10b. ABS Input Frequency Detection using the PTS and EPA Inputs**

```
;
; * Software Timer Interrupt *
;
EPAX_ISR:
         PUSHA                                   ; Save PSW
         LD      EPAIPV_Ptr,#EPA_PRIORITY; Read EPAIPV
         TIJMP   EPAIntX_JTBase_Ptr,EPAipv_Ptr,08H

Sw_Timer_ISR:
         AND     PTS_Select,#0FFE1h        ; Turn off PTS EPA interrupts
         LDB     WSR,#1FH
         ADD     COMP_TIME0_1F,#2500       ; Re-calculate COMP_0 and reset
         CLRB    WSR
         CLR     INV_SPEED_0+2             ; Clear Wheel Speed Registers
         CLRB    num_of_pulses_0+1         ; Clear pulse counter for new

Wheel_speed_Calc_0:
         NEGB    PTS_COUNT_EPA0                    ; Negate the Pulse Count
         SUB     INV_SPEED_0,FTIME_0,ITIME_0       ; get total period
         DIV     INV_SPEED_0,NUM_OF_PULSES_0       ; average Timer1 tick count
         Ld      Templ+2, #Speed_high_Constant     ; Load Templ with MAGIC number
         Ld      Templ, #Speed_low_Constant        ; to give units in Hertz.
         DIV     TEMPL, INV_SPEED_0                ; Result of wheel speed in Hertz.
         St      TEMPL, INV_SPEED_0                ; stored in INV_SPEED_0

         Call    Clear_RAM
         LD      EPAIPV_Ptr,#EPA_PRIORITY          ; Read EPAIPV
         TIJMP   EPAIntX_JTBase_Ptr,EPAipv_Ptr,08H

EPAIntX_Done:
         POPA
         RET

EPAIntX_JTBase:
         DCW     EPAIntX_Done    ;0
         DCW     Error_Loop      ;1
         DCW     Error_Loop      ;2
         DCW     Error_Loop      ;3
         DCW     Error_Loop      ;4
         DCW     Error_Loop      ;5
         DCW     Error_Loop      ;6
         DCW     Error_Loop      ;7
         DCW     Sw_timer_ISR    ;8

Error_Loop:
         SJMP    ERROR_LOOP
```

270873–43

**Program 10c. ABS Input Frequency Detection using the PTS and EPA Inputs**

## 7.5 EPA HSO Generation

Control over the generation of HSO, High Speed Output, is gained by the use of two SFRs, EPA_CONTROLn and EPA_TIMEn ("n" designates the number 0-9, of the EPA channel) for each output. The EPA_CONTROLn register controls the nature of the action to be taken when the EPA_TIMEn register matches the value in the specified time base register (i.e., either TIMER1 or TIMER2).

The event must be programmed by first writing to the EPA_CONTROLn register, and then to the EPA_TIMEn register. If the RE bit in the EPA_CONTROLn register is set, then the event programmed will occur every time a match between the time base and EPA_TIMEn occurs; otherwise the event will be disabled after the first match occurrence, and can be re-enabled by writing to the EPA_TIMEn register.

Note, Port1 must be configured for use by the EPA before any HSO can be generated. This is accomplished by writing a 1 to P1REG.n to turn off the pull-down, writing a 0 to the P1IO.n register to configure the pin as an output, and lastly, writing a 1 to P1SSEL.n bit to select P1.n for use by the EPA.

An interrupt will be generated each time a match occurs on an enabled channel (i.e., one for which EPA_CONTROL was written). Depending on the channel on which the interrupt occurs, it can be masked in one of two places: either in the core for channels 0-3, or in the EPA mask registers and the core EPAINTx bit for all other channels. See the EPA interrupt section for details.

### 7.5.1 SQUARE WAVE GENERATION

To generate a simple square wave output, the following code can be used. It first configures pin P1.1 for use with the EPA as a push/pull output. Next EPA channel 1 is configured to toggle its output, with automatic event re-enable, every time the value in Timer1 matches EPA_TIME1. EPA_TIME1 is initialized to be 3000h and Timer1 is set-up. Now, the EPA will automatically toggle EPA1 every time the value in Timer1 reaches 3000h.

```
; This program demonstrates the use of the EPA
; Compare function to produce a square wave
; output.  The output pin (EPA1) is toggled
; every time TIMER1 reaches 3000h.
;
; EPA  and SFR window definition for window 7B;
;
EA_CONTROL1_WEQU        0E4h:WORD
EPA_TIME1_W     EQU     0E6h:WORD
;
; port1 SFRs for window 7E
;
P1REG_W         EQU     0F4h:BYTE
P1SSEL_W        EQU     0F0h:BYTE
P1IO_W          EQU     0F2h:BYTE

        CSEG    AT      2080h

Ld      SP,#500h                ;set up stack
LdB     WSR,#7Eh
OrB     P1REG_W,#02h            ;force output high
AndB    P1IO_W,#0FDh            ;make P1.1 output
OrB     P1SSEL_W,#02h           ;make P1.1 for EPA

LdB     WSR,#7Bh
Ld      EPA_Control1_W,#078h            ;T1/togl/re-enable
Ld      EPA_TIME1_W,#3000h              ;action=3000h
LdB     70h,#0C2h
StB     70h,TIMER1_CONTROL[0]           ;count at 1us per

self:   Sjmp    Self
```

270873-44

A square wave with a duty cycle of other than 50% can be generated by using two channels in conjunction (EPA0 and EPA1, EPA2 and EPA3, COMP0 and EPA8, or COMP1 and EPA9). The following code generates two square waves, one with a 30% duty cycle, and the other with a 60% duty cycle. Both square waves have the same frequency.

First, the program configures Port 6 pins 0 and 1 for use by the EPA. Then the EPA channels are set up. For the 30% duty cycle wave, EPA8 is programmed to clear the output pin after 100 counts of TIMER1. COMP0 is programmed to set the output pin after 300 counts, and then reset the timer. Both channels have the re-enable bit set so the event will repeat automatically.

The 60% duty cycle is produced in a similar way, except that COMP1 is not programmed to reset the timer as this is being handled by COMP0.

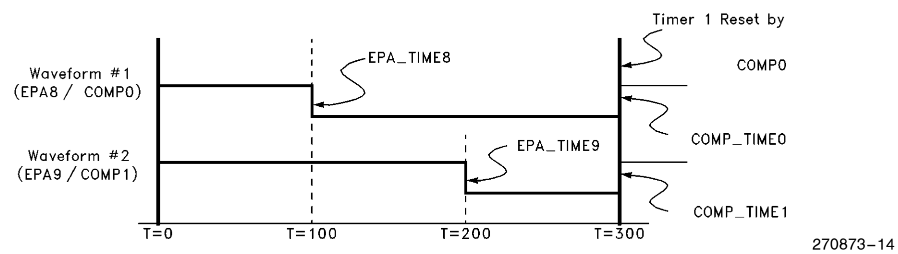



**Figure 7-6. Output Generated by Program 11**

```
;
;           This Program demonstrates the use of the EPA Compare function to
;           produce two PWM outputs.   No CPU OverHead, No PTS.
;
EPA_CONT8       EQU     0E0h    ;window to EPA_CONTROL8
EPA_TM8         EQU     0E2h    ;window to EPA_TIME8
EPA_CONT9       EQU     0E4h    ;window to EPA_CONTROL9
EPA_TM9         EQU     0E6h    ;window to EPA_TIME9
COMP_CONT0      EQU     0E8h    ;window to COMP_CONTROL0
COMP_TM0        EQU     0EAh    ;window to COMP_TIME0
COMP_CONT1      EQU     0ECh    ;window to COMP_CONTROL1
COMP_TM1        EQU     0EEh    ;window to COMP_TIME1
TIMER1_CON      EQU     0F8h    ;window to TIMER1_CONTROL
P6REG_W         EQU     0F5h    ;window to P6REG
P6SSEL_W        EQU     0F1h    ;window to P6SSEL
P6IO_W          EQU     0F3h    ;window to P6IO

CSEG    AT      2080h

        LdB     WSR,#7Eh
        OrB     P6REG_W,#03h      ;
        AndB    P6IO_W,#0FCh      ;P6.0 and P6.1 are outputs
        OrB     P6SSEL_W,#03h     ;P6.0 and P6.1 are special function

        LdB     WSR,#7Ch                  ;window 1F80-1F9f to E0-FF
        Ld      EPA_CONT8,#58h            ;reset pin, event re-enable
        Ld      COMP_CONT0,#69h           ;set pin, reset timer, event re-enable
        Ld      EPA_TM8,#100              ; 30% duty cycle
        Ld      COMP_TM0,#300             ;set after 300 counts, clear counter

        Ld      EPA_CONT9,#58h            ;reset pin, event re-enable
        Ld      EPA_TM9,#200              ; 60% duty cycle
        Ld      COMP_CONT1,#68h           ;set pin, event re-enable
        Ld      COMP_TM1,#300             ;same frequency as EPA8

        Ld      TIMER1_CON,#0C2h          ;enable timer, count at 1us per

Self:   Br      Self                      ;let EPA take over
```

270873–45

**Program 11. Generating 2 PWM Pulses Using No CPU Overhead**

### 7.5.2 PWM SIGNAL GENERATION WITHOUT PTS

Up to four PWM outputs can be generated in a manner similar to the square wave generation shown on the previous page.

One way to change the duty cycle would be to write a routine which monitors the state of the output. When it goes low, EPA_TIMEx register could be changed. If EPA_TIMEx was written when the clock was high, it would be possible for the duty cycle to become 100% for one cycle. (If EPA_TIMEx was written to a value less than TIMER1.)

Two nice things about using this method to generate a PWM is that it doesn't require any CPU or PTS overhead to maintain. And, any frequency and duty cycle can be produced with 16-bit resolution for both.

However, two EPA channels are being used for each PWM signal, and one dedicated timer is needed.

It is possible to generate a PWM signal using only one channel. Thus, up to 10 slower PWM signals can be generated. The code below demonstrates the method. For simplicity, only one PWM is produced.

The method used is similar to the HSO/CAM method used on other MCS-96 devices, such as the 8XC196KB. A wide range of duty cycles and frequencies can be produced. Only one timer is used.

The first segment of code sets up the registers used to hold the high (Const1) and low (Const2) time values of the PWM output. The next section configured PORT1.0 to be used by the EPA as an output. Following this, the EPA_CONTROL0 register is programmed. It is set up to toggle the output pin every time the value of TIMER1 matches EPA_TIME0.

The RE bit should NOT be set as the EPA_TIME0 register will be re-written after each edge, enabling the next event. The last part of the main loop starts TIMER1 running at 1us per count, and enables the EPA0 interrupt in the core.

The interrupt service routine is requested each time the EPA_TIME0 matches TIMER1. It checks the state of the PWM output (EPA0), to determine which value to add to the EPA_TIME0 register to set up for the next edge.

For example if the output is high, the value of Const1 is added to EPA_TIME0 and stored back into EPA_TIME0. The interrupt service routine will take 90 state times to set up a rising edge, plus an additional seven if the duty cycle has to be changed, and 79 states to set up a falling edge.

The duty cycle can be changed by modifying the values of Const1 and Const2. However, their sum must be kept the same to avoid changing the frequency. To change the duty cycle using the above code, the new values for the constants should be written to NConst1 and NConst2, followed by the setting of the "valid" flag.

In the previous example, it is assumed that the frequency is measured from rising edge to rising edge. So the values of the constants are changed by the interrupt service routine only after a rising edge was set up (i.e., the output is currently low). This insures that the frequency will not change momentarily.

```
; This program demonstrates the use of the EPA Compare function to
;   produce a PWM output.  The PTS is not used.
;
; EPA  and SFR window definition for window 7B
EPA_CONTROL0_W          EQU     0E0h:BYTE
EPA_TIME0_W             EQU     0E2h:WORD
EPA_CONTROL1_W          EQU     0E4h:WORD
EPA_TIME1_W             EQU     0E6h:WORD
;
; port1 SFRs for window 7E
P1SSEL_W                EQU     0F0h:BYTE
P1IO_W                  EQU     0F2h:BYTE
P1REG_W                 EQU     0F4h:BYTE
P1_0                    EQU     1FD4h:BYTE      ;P1 register
Const1                  EQU     060h:WORD       ;count for EPA0 high
Const2                  EQU     062h:WORD       ;count for EPA0 low
temp                    EQU     064h:BYTE       ;
NConst1                 EQU     066h:WORD       ;new value for EPA0 high
NConst2                 EQU     068h:WORD       ;new value for EPA0 low
Valid                   EQU     06Ah:BYTE       ;new data is valid

        CSEG    AT      2008h
        DCW     EPA0_ISR                        ;vector for EPA0 interrupt routine

        CSEG    AT      2080h
        DI
        DPTS
        Ld      SP,#500h                ;set up stack
        Ld      Const1,#100             ;20% duty cycle
        Ld      Const2,#400             ;2Khz freq.
        LdB     WSR,#7Eh
        OrB     P1REG_W,#01h            ;set P1.0 high
        AndB    P1IO_W,#0FEh            ;make P1.0 output
        OrB     P1SSEL_W,#01h           ;make P1.0 for EPA

        LdB     WSR,#7Bh
        Ld      EPA_Control0_W,#070h    ;Timer1, toggle output
        Ld      EPA_TIME0_W,Const1
        LdB     temp,#0C2h              ;enable timer, 1us
        StB     temp,TIMER1_CONTROL     ;start timer
        OrB     INT_MASK,#10h           ;Enable EPA0, interrupt
        ClrB    Valid                   ;no new data as of yet
        EI
self:   Sjmp    Self
;
;To change the duty cycle of the PWM output, Change the values of Const1 and Const2, but keep the
; sum the same or the frequency will change.  Then set VALID indicating new data for the PWM.
;
;Rising edge to rising edge defines the frequency, and so the data will only be changed after the
; rising edge set up.  This insures that the frequency will not change, momentarily, as a result
; of changing the duty cycle.
;
EPA0_ISR:
        PushA
        Ldb     WSR,#7Eh
        JbS     0F4h,0,setf                     ;read p1.0 thru window
        Add     temp,Const2,EPA_TIME0[0]        ;set up rising edge
        JbC     valid,0,out                     ; if no new data, exit
        Ld      Const1,NConst1                  ;update const1
        Ld      Const2,NConst2                  ;update const2
        ClrB    valid                           ;note data was updated
        sjmp    out
setf:   Add     temp,Const1,EPA_TIME0[0]        ;set up falling edge
out:    St      temp,EPA_TIME0[0]
        POPA
        RET
```

270873–46

**Program 12. PWM Generation Using Interrupts**

### 7.5.3 PWM GENERATION WITH PTS

The PTS has two modes which can be used to generate PWM signals: PWM (up to 2 PWMs) and PWM TOGGLE (up to 4 PWMs). The latter uses the same method as shown on the previous page, but the PTS instead of a ISR handles updating the EPA_TIME0 register.

Program 13 produces the same results, but uses the PTS in PWM TOGGLE mode. The first block of code initializes the PTS control block. The PWM toggle mode is used, with the source registering pointing to the EPA_TIME0 register. Const1 and Const2 are chosen so that a 2KHz waveform with a 20% duty cycle is produced.

PORT1 is then configured, EPA channel 0 is programmed, and TIMER1 is started. *It is important to note that EPA0 is forced low. This insures that the proper polarity PWM is generated. The EPA will just toggle the output, not caring what the initial state was.*

EPAINT0 is enabled in both the core and the PTS_SELECT register. This allows the PTS to continuously set up the edges, instead of having the CPU handle it.

*However, due to a bug in the interrupt handler on A-Step silicon, an interrupt service routine for EPAINT0 is still needed. If a PTS interrupt should occur within the latency time of the starting of another normal interrupt, the PTS interrupt will NOT be serviced, but rather a call will be made to the interrupt routine corresponding to the PTS interrupt. Therefore, the interrupt service routine for EPAINT0 should set the interrupt pending bit for EPAINT0 and exit. This will force a call to the PTS service routine.*

Note that the PTS routine only takes 15 states to execute; this is more than an 80% reduction in the time needed to maintain a PWM output.

The PWM Toggle PTS cycle operates as follows:

1) The value pointed to by PTS_SOURCE is read.
2) Const1 or Const2 is added to this value, depending on the state of TBIT in PTS_CONTROL. TBIT = 0 selects Const1
3) The result is stored back into the value pointed to by PTS_SOURCE. The TBIT is then toggled.

The duty cycle can be changed in the program by first writing the new values for PTS_CONST1 and PTS_CONST2 into NCONST1 and NCONST2, respectively. Then the PTS should be disabled from servicing EPA0 interrupts.

The EPA0 ISR will change the duty cycle. If the output pin is in the high state when it is called, it will perform a "manual" PTS cycle to set up the falling edge.

When it is called and the output is in a low state, it will update the PTS constants and re-enable the PTS for EPA0. Lastly, it will force a call to the PTS by setting the INT_PEND bit for EPA0.

Program 14's methods work well for generating PWM output as long as there is enough time between edges for either the CPU or the PTS to set up the next edge. However, if two edges are placed very close together, there will not be enough time to set up the second edge.

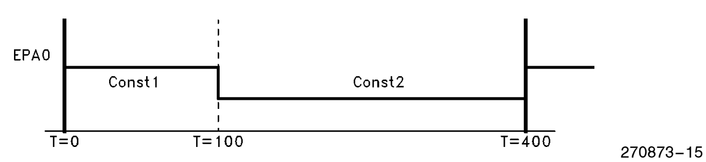



**Figure 7-7. Output of Program 12 and 13**

```
; This program demonstrates the use of the EPA Compare function with the
; PTS toggle mode to produce a PWM output.
; EPA  and SFR window definition for window 7B;
    EPA_CONTROL0_W          EQU     0E0h:BYTE
    EPA_TIME0_W             EQU     0E2h:WORD
; Port1 SFRs used with window 7E
    P1SSEL_W                EQU     0F0h:BYTE
    P1IO_W                  EQU     0F2h:BYTE
    P1REG_W                 EQU     0F4h:BYTE
    P1PIN_W                 EQU     0F6h:BYTE
            CSEG    AT      2048h
    DCW     PTS_UNUSED              ;PTS interrupt for EPA0
            CSEG    AT      2008h
    DCW     EPA0_ISR                ;vector for EPA0 interrupt routine
            RSEG    AT      070h    ;define pts control block
    PTS_UNUSED:     DSB     1
    PTS_CON:        DSB     1
    PTS_SRC:        DSW     1
    PTS_CONST1:     DSW     1
    PTS_CONST2:     DSW     1
    NConst1:        DSW     1
    NConst2:        DSW     1
            CSEG    AT      2080h
START:      DI
            DPTS
            Ld      SP,#500h        ;set up stack
            Ldb     PTS_CON,#43h    ;PWM toggle mode, CONST1 First
            Ld      PTS_SRC,#EPA_TIME0      ;source is epa_time0
            Ld      PTS_CONST1,#100         ;on for 100 us
            Ld      PTS_CONST2,#400         ;off for 400 us
            LdB     WSR,#7Eh
            OrB     P1REG_W,#01h            ;force P1.0 high
            OrB     P1SSEL_W,#01h           ;make P1.0 for EPA
            AndB    P1IO_W,#0FEh            ;make P1.0 output
            LdB     WSR,#7Bh
            Ld      EPA_Control0_W,#070h    ;Timer1, toggle output
            Ld      EPA_TIME0_W,PTS_CONST1
            LdB     70h,#0C2h               ;enable timer,1us
            StB     70h,TIMER1_CONTROL      ;start timer
            OrB     INT_MASK,#10h           ;Enable EPA0 interrupts
            OrB     PTS_SELECT,#10h         ;Enable EPA0 to PTS interrupts
            EPTS
            EI
    self:   Sjmp    Self
    ;To change the duty cycle, load the new values of the constants into NConst1 and NConst2. Then
    ; disable PTS servicing of the EPA0 interrupt. The EPA_ISR will handle the task of changing the
    ; duty cycle. If needed (if the output is high) it will first perform a manual PTS cycle.
    EPA0_ISR:
            PUSHA
            JbS     PTS_SELECT,4,abort      ;check for bug
            LdB     WSR,#7Eh
            JbS     P1PIN_W,0,change
            Ld      PTS_CONST1,NConst1      ;change duty cycle
            Ld      PTS_CONST2,NConst2
            LdB     PTS_CON,#42h            ;set TBIT to proper value
            OrB     PTS_SELECT,#10h         ;restart PTS
    abort:  OrB     INT_PEND,#10h           ;go to the correct place
            POPA
            RET
    change: LdB     WSR,#7Bh                ;set up falling edge
            Add     EPA_TIME0_W,PTS_CONST1  ;do manual PTS cycle
            POPA                            ;next call to this routine
            RET                             ;will change duty cycle.
```

270873–47

**Program 13. Generate a PWM on EPA0 using the PTS Toggle Mode**

```
; EPA and SFR window definition for window 7B
    EPA_CONTROL0_W            EQU     0E0h:BYTE
    EPA_TIME0_W               EQU     0E2h:WORD
    EPA_CONTROL1_W            EQU     0E4h:WORD
    EPA_TIME1_W               EQU     0E6h:WORD
; port1 SFRs for window 7E
    P1SSEL_W                  EQU     0F0h:BYTE
    P1IO_W                    EQU     0F2h:BYTE
    temp                      EQU     060h:WORD
            CSEG    AT        2048h
    DCW     PTS_UNUSED0       ;PTS CONTROL BLOCK for EPA0
            CSEG    AT        2046h
    DCW     PTS_UNUSED1       ;PTS CONTROL BLOCK FOR EPA1
            CSEG    AT        2008h
    DCW     EPA0_ISR          ;vector for EPA0 interrupt routine
            CSEG    AT        2006h
    DCW     EPA1_ISR
            RSEG    AT        078h                  ;define pts control block epa0
    PTS_UNUSED0:              DSB     1
    PTS_CON0:                 DSB     1
    PTS_SRC0:                 DSW     1
    PTS_CONST0:               DSW     1

            RSEG    AT        070h                  ;define pts control block epa1
    PTS_UNUSED1:              DSB     1
    PTS_CON1:                 DSB     1
    PTS_SRC1:                 DSW     1
    PTS_CONST1:               DSW     1

            CSEG    AT        2080h
            DI
            DPTS
            Ld      SP,#500h                        ;set up stack
            LdB     PTS_CON0,#40h                   ;PWM  mode
            Ld      PTS_SRC0,#EPA_TIME0             ;source is epa_time
            Ld      PTS_CONST0,#150                 ;set every 150us
            LdB     PTS_CON1,#40h                   ;PWM mode
            Ld      PTS_SRC1,#EPA_TIME1
            Ld      PTS_CONST1,#150                 ;clear every 150us
            LdB     WSR,#7Eh
            OrB     P1REG_W,#01h                    ;make P1.1 high
            AndB    P1IO_W,W,#0FDh                  ;make P1.1 output
            OrB     P1SSEL_W,#02h                   ;make P1.1 for EPA
            LdB     WSR,#7Bh
            Ld      EPA_Control0_W,#060h            ;Timer1, set output
            Ld      EPA_TIME0_W,#0000
            Ld      EPA_Control1_W,#150h            ;Timer1, clear output
            Ld      EPA_TIME1_W,#0050               ;pulse for 50us
            LdB     temp,#0C2h                      ;enable timer, 1us
            StB     temp,TIMER1_CONTROL             ;start timer
            OrB     INT_MASK,#18h                   ;Enable EPA0,EPA1 interrupts
            OrB     PTS_SELECT,#18h                 ;Enable EPA0,EPA1 to PTS interrupts
            EPTS
            EI
    self:   Sjmp    Self

    EPA0_ISR:
    ;The only way we can get here is from the PTS bug.  Revector to the PTS routine
            OrB     INT_PEND,#10h                   ;go to the correct place
            RET
    EPA1_ISR:
    ;The only way we can get here is from the PTS bug.  Revector to the PTS routine
            OrB     INT_PEND,#08h                   ;go to the correct place
            RET
```

270873–48

**Program 14. Generate a PWM Using the PTS PWM Mode and the Re-Map Feature**

The PWM mode of the PTS can be used to work around this. Two EPA channels are used together (either EPA0 and EPA1 or EPA2 and EPA3). (See Program 14) One channel will control the rising edge, and the other will control the falling edge of the output. Thus, there is no need for the PTS or the CPU to intervene between the edges, allowing them to be placed closer together. However, there must be time to set up two edges before another edge can occur. Program 14 demonstrates this mode with the PTS.

The code is similar to what was presented above, but there are a few differences to be pointed out. First, there are two PTS control blocks. One controls the time from rising edge to rising edge, and the other controls the time from falling edge to falling edge. Normally, these should be the same.

The frequency of the PWM wave is controlled by the constants in the two PTS control blocks, and the frequency of TIMER1. The duty cycle is controlled by the difference of the two time registers, EPA__TIME0 and EPA__TIME1. For this example the pulse will be high for 50$\mu$s.

The duty cycle can be changed in a manner similar to that which was used in the previous PTS example. Note, however, that only EPA1 needs to be disconnected from the PTS, as it controls the falling edge.

Another thing to note is the manner in which the two EPA channels are configured. EPA0 is set up to force EPA1 high when EPA__TIME0 matches TIMER1, while EPA1 is set up to clear EPA1 when EPA__TIME1 matches TIMER1. Also, the 8th bit in EPA__CONTROL1 must be set to a 1. This is to allow EPA__CONTROL0 to control EPA1.

*A word of caution is needed here: do NOT set both EPA__TIME0 and EPA__TIME1 to the same value and expect to get a 0% duty cycle PWM. This will not happen due to the fact that when there is a conflict between EPA commands (i.e., set and clear the pin at the same time), the EPA will toggle the pin at the EPA__TIME value.*

Finally, note that the PWM output appears at EPA1 (or EPA3 if EPA2 and EPA3 are working together). Thus, PORT1 pin 1 is configured as an output for the EPA. Pin 1.0 can still be used as an LSIO pin.

### 7.5.4 PWM GENERATION USING SOFTWARE

As a final example Program 15, creating an PWM output on channel 9, will be considered. Since only channels 0-3 have direct interrupt lines to the core and PTS, this example will also demonstrate the use of the TIJMP instruction. The PTS cannot be used with channels 4-9 since there is only one bit in the PTS__SELECT register for all 5 channels. The PTS cannot determine which channel caused the interrupt, and is therefore unable to modify the proper EPA__TIMEn register.

*NOTE: Parts of the jump vector table were left out. They all contain a jump to the error routine.*

This program is very similar to the first PWM example, except that a different EPA channel is used. Since EPA9 is used, and it doesn't have a direct interrupt line to the CPU, the following changes have been made. First the EPAINT9 bit in the EPA__MASK register has been set, allowing EPA9 interrupts to occur. The EPAINTx flag in the INT__MASK has also been set.

```
$Include (Macro.KR)
;
;  This program demonstrates the use of the EPA Compare function to produce a PWM output.  The
;   TIJMP instruction is also demonstrated.
;
; EPA  and SFR window definition for window 7C;
;
    EPA_CONTROL9_W          EQU     0E4h:BYTE
    EPA_TIME9_W             EQU     0E6h:WORD
; port1 SFRs for window 7E
    P6SSEL_W                EQU     0F1h:BYTE
    P6IO_W                  EQU     0F3h:BYTE
    P6_0                    EQU     1FD5h:BYTE          ;P6 register
    Const1                  EQU     060h:WORD           ;count for EPA9 high
    Const2                  EQU     062h:WORD           ;count for EPA9 low
    temp                    EQU     064h:BYTE
    EPAIPV_PTR              EQU     070h:WORD           ;pointer to EPAIPV SFR
    EPA_TI_BASE             EQU     072h:WORD           ;point to base of TIJMP table
            CSEG    AT      2000h
            DCW             EPAX_ISR                    ;vector for EPAX interrupt routine

            CSEG    AT      2080h
            DI
            DPTS
            Ld      SP,#500h                    ;set up stack
            Ld      Const1,#100                 ;20% duty cycle
            Ld      Const2,#400                 ;2Khz freq.
            LdB     WSR,#7Eh
            AndB    P6IO_W,#0FDh                ;make P6.1 output
            OrB     P6SSEL_W,#02h               ;make P6.1 for EPA9
            LdB     WSR,#7Ch
            Ld      EPA_Control9_W,#070h        ;Timer1, toggle output
            Ld      EPA_TIME9_W,#00
            LdB     temp,#0C2h                  ;enable timer, 1us
            StB     temp,TIMER1_CONTROL         ;start timer
            OrB     INT_MASK,#01h               ;Enable EPAX, interrupt
            Ld      EPAIPV_PTR,EPA_MASK[0]      ;use EPAIPV_PTR as temp
            Or      EPAIPV_PTR,#0400h           ;Enable EPA INT9
            St      EPAIPV_PTR,EPA_MASK[0]
            EI
    self:   Sjmp    Self
    EPAX_ISR:
            PushA
            Ld      EPAIPV_PTR,#1FA8h           ;set pointer to EPAIPV SFR
            Ld      EPA_TI_BASE,#JUMP_TABLE1    ;pointer to jump table
            TiJmp   EPA_TI_BASE,EPAIPV_PTR,07Fh
    ; other EPA ISRs go here
    EPAINT9_1:
            Ldb     WSR,#7Eh
            JbS     0F5h,1,setf                 ;read p6.1 thru window
            Add     temp,Const2,EPA_TIME9[0]    ;set up rising edge
            sjmp    out
    setf:   Add     temp,Const1,EPA_TIME9[0]    ;set up falling edge
    out:    St      temp,EPA_TIME9[0]
            TiJmp   EPA_TI_BASE,EPAIPV_PTR,07Fh ;handle next interrupt
     EXIT1:
            POPA
            RET
    ;Trap any other interrupt, non should have occurred.

    error:          Sjmp    error
```

270873–49

**Program 15a. Generate a PWM Output Using EPA9 and Software Interrupts**

```
; Following is the table of entry points for the EPA ISRs. Note that there are actually two tables
;  which are interleaved. This is because there is a small problem with the TIJMP/EPAIPV. The
;  EPAIPV increments in steps of two and the TIJMP multiplies the offset by two. Thus the final offset
;  into the table is always a multiple of 4 and not 2 as one would expect.
JUMP_TABLE1:
        DCW         EXIT1
JUMP_TABLE2:
        DCW         error           ; EXIT2
        DCW         error           ; Timer2_over1
        DCW         error           ; Timer2_over2
        ...         .....
        DCW         error           ; OVRINT1_1
        DCW         error           ; OVRINT1_2
        DCW         error           ; OVRINT0_1
        DCW         error           ; OVRINT0_2
        DCW         EPAINT9_1       ; EPAINT9_1
        DCW         error           ; EPAINT9_2
        ...         .....
        DCW         error           ; EPAINT4_1
        DCW         error           ; EPAINT4_2
END
```

270873–50

**Program 15b. Generate a PWM Output Using EPA9 and Software Interrupts**

The interrupt service routine has also changed, since now the exact source of the interrupt must be determined in software. A TIJMP instruction is used to minimize overhead. The TIJMP table contains pointers to the various interrupt service routines. Actually, the table consists of two interleaved tables due to the way in which TIJMP and the EPAIPV register work together: The EPAIPV vector always is a multiple of 2, and the TIJMP instruction multiplies by 2 to calculate the offset into the table. Thus offsets into the jump table will always be a multiple of 4. (This may change on latter parts.)

The TIJMP instruction takes three arguments: base location of the table, an offset into the table, and a mask for the offset. The mask for the offset can be used to change the interrupt priorities, but does nothing in the example. The EPAIPV (EPA Interrupt Priority Vector) always contains a value indicating the highest priority interrupt that is pending (and masked). This register should be read until it returns a 00h, indicating that all interrupts have been processed. This is the only way to clear the EPA__PEND and EPA__PEND1 register and INT__PEND bit 0 in the core. The above example does this by ending each EPA interrupt source's service routine with another TIJMP instruction. When the EPAIPV returns a 00h, flow is handed over to the EXIT subroutine, which handles exiting from the EPAINTx service routine.

## 7.6 Top 5 Issues with the EPA

(1) Read the EPA__TIME register after each EPA (input capture) interrupt.

(2) All EPA__CONTROL and COMP__CONTROL registers should be written as WORD. This will make the users code compatible with future KX devices. Reserved bits are written to zero.

(3) PWM Generation can be accomplished via:

1. 10 PWMs with Software Interrupts (EPA0–EPA9)
2. 4 PWMs with PTS PWM Toggle Mode (EPA0–EPA3).
3. 2 PWMs with PTS PWM Mode and re-mapping outputs (EPA1 and EPA3)
4. 4 PWMs using a dedicated Timer with Re-mapping (EPA1, EPA3, EPA8 and EPA9).

(4) Before exiting an EPAINTx interrupt service routine, the EPAIPV register MUST be read until it equals "00".

(5) Due to a Bug in A-step Silicon, the EPA__MASK1 and EPA__PEND1 must be written as words.